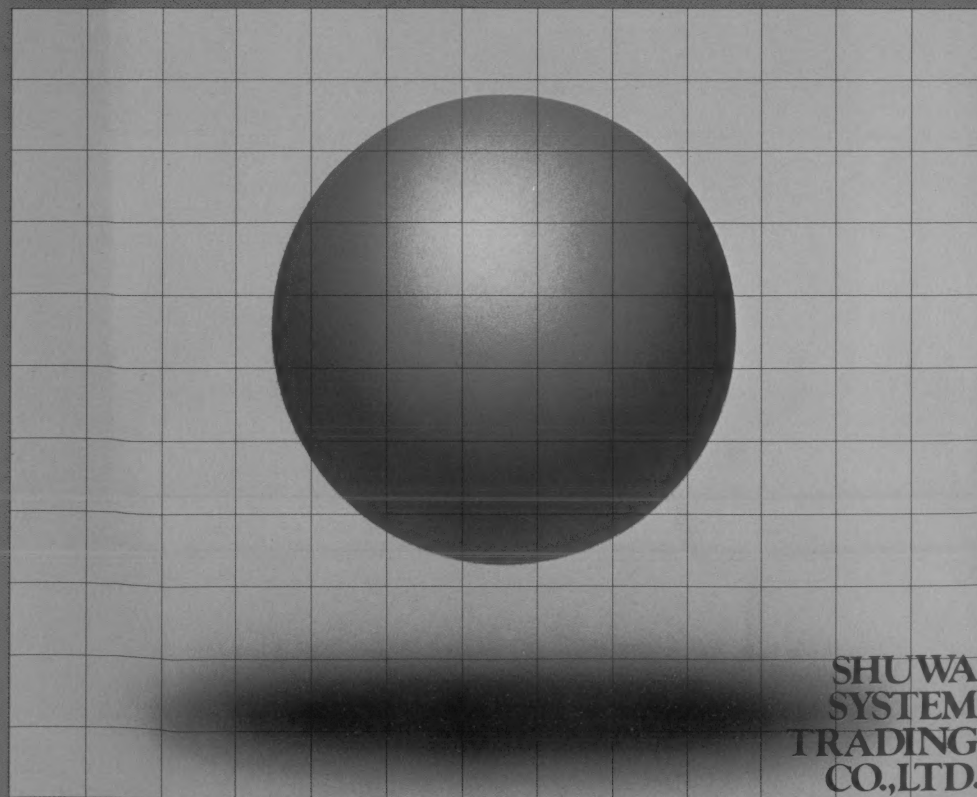


ザ・プロテクトII (プログラム解読法入門)

PC-9800シリーズ  
ザ・プロテクトII  
プログラム解読法入門  
井上智博・技術開発室



秀和システムトレーディング株式会社 定価 2,200円

ISBN4-87966-116-3 C3055 ¥2200E

新刊書籍発売中

Microsoft C Ver 3.0  
ユーザーズガイド

MS-Cの操作法から言語仕様、ライブラリ関数の使い方など、そのすべてを詳細に解説しユーザの便に供する。

A5判 定価3,200円 送料300円

Microsoft C Ver 3.0  
テクニカルガイド

C言語を駆使してMS-DOSの機能を十分に引き出すための、ユーザに耳よりなテクニックを紹介。

A5判 定価3,200円 送料300円

PC-9800シリーズ

BASICユーティリティブック

DISK-BASIC上で動作する各種ユーティリティを満載し、全98ユーザの便宜をはかるプログラム集。

B5判 定価2,800円 送料300円

PC-9800シリーズ

MS-DOSユーティリティブック

C言語やアセンブラによる各種ユーティリティプログラムを掲載し、バージョンアップ時のアドバイスなど、技術の向上にも役立つ。

B5判 定価2,800円 送料300円

PC-9801 VM/UV/VX/LT

テクニカルガイド

PC-98 新機種をフルに使いこなすために、ハード、ソフトの両面からその利用法を徹底的に解説。

B5判 定価2,800円 送料300円

## PC-9800シリーズ”売行良好書のご案内

- **ザ・プロテクト**  
A5判 定価1,900円 送料300円 別売ディスクパック版 定価5,800円 送料300円
- **ユーティリティプログラムMS-DOS I コマンドアプリケーション**  
B5判 定価2,900円 送料300円 別売ディスクパック版 定価8,800円 送料300円
- **ユーティリティプログラムMS-DOS II システムアプリケーション**  
B5判 定価2,900円 送料300円 別売ディスクパック版 定価8,800円 送料300円
- **ユーティリティプログラム応用実例集**  
B5判 定価2,500円 送料300円 別売ディスクパック版 定価8,800円 送料300円
- **N88-日本語BASIC(86) インタプリタとコンパイラ**  
B5判 定価2,800円 送料300円 別売サービス版 定価8,800円 送料300円

ザ・プロテクトII

秀和システムトレーディング株式会社 東京都港区赤坂8-5-29 チッカールビル

プロテクトの読み方と解読法入門



# 究極のプロテクトは可能か！

限界といわれて久しいプロテクト技術誂一つの回答を示し、プロテクト技術の究のすがたを探る。

定価=2,200円

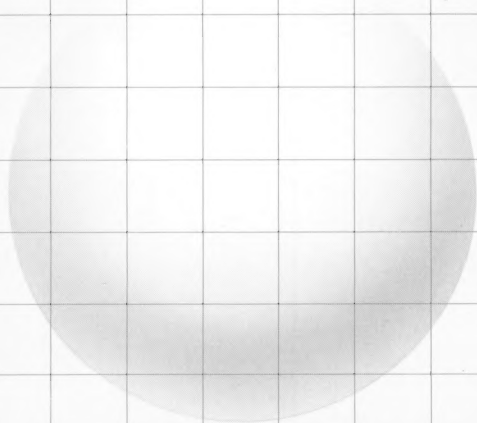
PC-9800シリーズ

# ザ・プロテクトII

## プログラム解読法入門

井上 智博・技術開発室

# The Protect II



SHUWA  
SYSTEM  
TRADING  
CO.,LTD.

■注意

- 〔1〕本書は筆者等が独自に調査した結果を出版したものです。
- 〔2〕本書は内容について万全を期して作製いたしましたが、万一、ご不審な点や誤り、記載もれなどお気付きの点がありましたら、出版元まで書面にてご連絡ください。
- 〔3〕本書の内容に関して運用した結果の影響につきましては、上記〔2〕項にかかわらず責任を負いかねます。ご了承ください。
- 〔4〕本書の全部または一部について、出版元から文書による許諾を得ずに複製することは禁じられています。

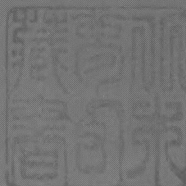


PC-9800シリーズ

# ザ・プロテクトII

## プログラム解読法入門

井上智博・技術開発室



秀和システムトレーディング株式会社

# まえがきにかえて

## ■どうしても複製がとれないとき

プロテクトを取り巻く情勢は刻々と変化しています。前巻『PC-9800 シリーズ ザ・プロテクト』が発行されたのが1986年の11月でした。もうそれから1年以上が経過していることになります。プロテクトという言葉自体もさまざまな意味に変化しました。プロテクトに対する考え方もずいぶんと変わりました。

本書の主題に入る前に、まずプロテクトということがらについて、もう一度見つめ直してみようと思います。

## ■“究極のプロテクト”は本当に究極か

“究極のプロテクト”すなわち、絶対に破れないプロテクトは可能か？ この答えは皆さんでお出しください、というのが前作の回答でした。新しいプロテクトが開発されれば、それは常にその時代の究極でした。しかし、その新しいプロテクトに対抗するための手段が考案されると、それは究極ではなくなってしまいます。ちょうど“いたちごっこ”にたとえられるように、強力なプロテクトが開発されれば、それを上回る強力なコピーツールが開発され、それに対抗するべく、さらに強力なプロテクトが開発される……といった具合です。こう考えると、この“いたちごっこ”は、プロテクトとコピーツールの歴史といってもよいでしょう。この歴史を振り返ることによって、その時代時代での“究極のプロテクト”がわかるわけです。

“究極のプロテクト”とは、絶対にコピーのとれないプロテクトをいいます。4、5年前まではプロテクトそのものの存在が希少で、たとえあっても、それはとるにたらないものばかりでした。それでも当時としては“究極”であったのです。

たとえば、次のようなものです。

まず、フロッピーディスクをBASICでフォーマットします。そこにソフトウェアを順々に格納していくわけですが、よほど大きなソフトウェアでない限り、フロッピーディスク上には空きが生じます。

その空き領域へは、フォーマットしてから書き込みが行われていないわけですから、フォーマットしたままのデータが入っているはずです（BASIC の場合、ふつう FFH）。しかし、このプロテクトでは、ここを 00H などと異なる値にしておくのです。backup.n88 などを用いてふつうにコピーをとれば、この部分まではコピーされませんから（backup.n88 は、ファイルが存在するトラックのみをコピーするように設計されている）、チェックすればオリジナルが複製かがわかるわけです。具体的には、この部分は最終トラック（2D であったので 79）の最終セクタ（16）に設定されていました。これを BASIC から DSKI\$を用いて読んでいたのです。

```
A$=DSKI$(1,1,39,16):IF A$<>STRING$(255,0) THEN  
NEW
```

プログラムを解説し、このような内容の行を発見できなければ、なぜプログラムが停止し、リストが消えてしまうのかわからないわけです。この方法は、backup.n88 や xfiles.n88 などの標準的なコピープログラムに対して有効でした。一般のユーザに対しては、この程度の水準で十分だったのです。

その後、セクタ長を変化させるというプロテクトが出現しました。その目的は、コピープログラムの動作を停止させることにありました。backup.n88 などでは、BASIC の標準的なフォーマットを対象としていたため、コピーできないトラックが存在すると、そこで動作を停止してしまうのです。このようなタイプのプロテクトが出現したあたりから、コピーツールと呼ばれるソフトウェアが、徐々に姿を現しはじめました。

それからは、FDC（フロッピディスクコントローラ）の規格外のフォーマットを用いたり、特殊な装置を用いてフォーマットを行ったりと、より複雑で難解なプロテクトが考案されてきます。最も高度なプロテクトの類に、フォーマット時の回転数（または書き込みクロック）を変化させるというものがありますが、これもアナログコピーを行うことによってコピーすることができます。現在では、フロッピディスク上のフォーマットを読み取るのではなく、電気信号の変化を読み取る段階にまできているのです。

## ■プロテクト成立のための条件

さて、いろいろなプロテクトが多数存在しますが、プロテクトの最終目的はいったい何なのでしょう。それは、いうまでもなく“コピーが使えない”ようにすることです。ところで、この“コピーが使えない”ようにするためには、どのような条件が満たされなければならないのでしょうか。逆にいえば、どの条件をなくせばプロテクトの効力がなくなるのでしょうか。もう一度見直してみましょう。

プロテクト成立のための条件は、まずコピーできないということです。コピーができなければ、複製品そのものが作成できないわけです。これはプロテクトの基本ともいえます。

ここでコピーできないとは、

- ・標準的なバックアッププログラム  
(backup.n88, DISKCOPY コマンドなど)
- ・市販されるプロテクトを対象としたコピーツール

によってコピーされないことをいいます。前者は、自ら動作するOSのファイルフォーマットに逆らうものがコピーできないのに対し、後者は、あらかじめそのようなものにも対応するかたちで存在しています。前者に対しては、前述したセクタ長を変化させるなどの単純な手段で対抗できますが、後者では、そのようなものは当たり前ですので、もっと複雑な手段が必要となってきます。

コピーツールによって、コピーすることのできないプロテクトについては、既刊『PC-9800 シリーズ ザ・プロテクト』を参照してください。

プロテクト成立のための、次の条件は“コピーされたものが動作しない”ということです。こうなると、コピーには成功したように見えても、いざ動作させると動かないという事態が発生するわけです。実際には、ソフトウェアが実行されている途中でフロッピーディスクのフォーマットをチェックし、オリジナルであるかどうかを調べればよいのです。中には、コピーツールでコピーされてしまうように見えるものもあります。したがって、フォーマットが正常であるかどうかを確認することは、絶対に必要なことなのです。

プロテクト成立のためには、この2つの条件の満たされることが必要です。ここで示した条件はかなりおおざっぱなもので、詳しく突き詰めればもっと多様な条件が出てくるはずですが、しかし、おおかたはこのようなところです。プロテクトとは、さまざまな手段を講じて、この条件を満たすためのものなのです。

## ■コピーツールの方法

コピーツールの役割りは、プロテクト成立のための条件を満たさせないようにすることです。

1番目の条件に対しては、コピーツール自体がさまざまなフォーマットを判別でき、かつ、それを作成することのできることが条件となります。すなわち、どのようなフォーマットが現れても、それを正確に判別できて、それと同じフォーマットを再現できることが必要なのです。従来は、コピーツールといえはこの能力を最も重視しました。

しかしここで問題が出てきます。それは正しくコピーされたかどうかはコピーツールにはわからないということです。正しくコピーされていないければ、いくらコピーが正常に終了しても、プログラムの実行中に、オリジナルかどうかのチェックが行われてしまえば、そこでおしまいになります。

そこでコピーツールでは、2番目の条件を満たさせないようにする必要があります。同じフォーマットが作成できない場合に、オリジナルかどうかのチェックを行わせないようにするのです。

オリジナルかどうかのチェックを行うのはソフトウェア側の勝手ですが、コピーツール側では防ぎようがないと思われてきました。そしてコピーツール側では、ソフトウェアを手作業によって解読して、チェックを行っている場所を捜し出して操作し、常にオリジナルであるという回答を出すようにしてきました。これをファイラーとか、パラメータと呼んでいます（コピーツールにより呼び方が異なる）。

これらはソフトウェアに個別に対応していて、ソフトウェアにかけられているプロテクトを無効にする働きをもっています。一度この方式でコピーしてしまえば、あとは標準的なバックアッププログラムによりコピーすることが可能になります。



## ■もう一つのプロテクト

フロッピディスクのフォーマットを変化させるといった、ハードウェアによるプロテクトは、もはや行くつくところまで行ったというのが一般的な見方です。では、次なるプロテクトとはと聞かれば、それはソフトウェアによるプロテクトだといわざるを得ません。皆さんは、ソフトウェアでプロテクトをかけることができるのか、それは破られないのか、という疑問を抱くかもしれません。しかし、それができるのです。ハードウェアによるプロテクトとは比較にならないほど多様なパターンを持ち、複雑で難解なものが実現できるのです。本書では、“もう一つのプロテクト”と題してこれを中心に取り上げます。



## 本書を読む前に

### ○プログラム例について

本書には多くのプログラム例が掲載されていますが、アセンブリ言語によるものは、MS-DOS 付属のマクロアセンブラ (MASM version 1.27,3.00) に準拠し、C 言語によるプログラムは Lattice C version J3.10 (version 3.00 でも可能) に準拠したものです (Lattice C は別売)。ただし、プログラムは断片的なものであり、そのままのかたちでアセンブラ、あるいはコンパイラにかけることはできません。

また、プリンタへの出力例は、特にことわりのないかぎりは SYMDEB によるものです。なお、SYMDEB 独自の機能を使用することは極力避けました。DEBUG においても、ほとんどが適用可能です。

### ○ユーティリティについて

本書には多くのユーティリティが掲載されており、アセンブリ言語で記述されています。そして、そのすべてが MS-DOS 上で動作するものです。ここに掲載されるユーティリティは、ソースリストのかたちをなしていますから、実際に実行するにはアセンブルして、実行ファイルの形式にしなければなりません。実行ファイルにするには、ユーティリティによって少し異なる方法をとらなければなりません。

ユーティリティ名の拡張子が“.EXE”である場合には、次の手順でアセンブルし、実行型ファイルを作成してください。このとき、MASM.EXE、LINK.EXE とソースファイルが同じディレクトリにあるものとします。また、ソースファイルの名前が“SOURCE.ASM”であるものとします。

```
MASM SOURCE;  
LINK SOURCE;
```

ユーティリティ名の拡張子が“.COM”である場合には、上記の手順にさらに以下の手順を追加してください。

EXE2BIN SOURCE SOURCE.COM

同様に、EXE2BIN.EXE が同じディレクトリになくてもなりません。不必要なファイルが残るようであれば DEL コマンドによって削除してください。

### ○アドレスの表現について

本書では、アドレスの表現を以下のように統一しました。

SSSS : OOOO  
AAAAA

SSSS はセグメントの値で、OOOO はオフセットの値です。またセグメントとオフセットによる表現のほかに、AAAAA という絶対表現も部分的に採用しています。数値は、16 進数であることを明示するために、末尾に”H”サフィックスを付加しています。

### ○使用機器構成について

本書の執筆にあたって使用した機器の構成は、以下のとおりです。

#### ●本体：

PC-9801M2 (8086CPU) / PC-9801VX2 (V30,80286CPU)

#### ●実装メモリサイズ：

640KB (PC-9801M3) / 640KB (PC-9801VX2)

#### ●DISK BASIC システムディスク：

PC-98H43-MW (K) (PC-9801M2)

PC-98H47-MW (K) (PC-9801VX2)

#### ●MS-DOS システムディスク：

PS98-121-HMW / PS98-125-HMW

## プログラム解読法入門

## 序

まえがきにかえて 3

本書を読む前に 8

## 基礎編

## 1

1 解読にあたって 17

1.1 なぜ解読が必要か 17

1.2 解読にのぞむ 19

2 解読の定石 29

2.1 ファイルを調べよう 29

2.2 まずは比較検討を 37

2.3 INT 1BH発見がポイント 38

2.4 環境変数を調べる 40

## 2

## 3

3 解読支援ツール 41

3.1 実行中断・レジスタ値表示 41

3.2 ディスクアクセスのロギング 51

3.3 システムコールのロギング 62

## 応用編 I

## 1

- 1 プロテクト表現のテクニック 65
  - 1.1 チェック→エラーは早すぎる 65
  - 1.2 エラーを出すだけでは能がない 69
  - 1.3 プロテクトの方法は1つではない 74

## 2 プログラムを読みにくくする 83

- 2.1 意味のない命令を多用する 83
- 2.2 定石をあえて破る 92
- 2.3 意味のない分岐 99
- 2.4 未定義命令を使う 100
- 2.5 ソフトウェア割り込みを使う 103
- 2.6 ハードウェアを頻繁にアクセス 107

## 2

## 3

- 3 目立つ命令をかくす 109
  - 3.1 INT 命令をかくす 109
  - 3.2 IN/OUT 命令をかくす 115

## 4 MS-DOS版プロテクト技法 117

- 4.1 不良クラスタを作る 117
- 4.2 ダミーファイルを作る 129
- 4.3 チェックの方法 131

## 4

## 応用編 II

## 1

- 1 ツールに対抗する 135
  - 1.1 ツールの命を無効にする 135
  - 1.2 親をチェックする 146
  - 1.3 実行時間をチェックする 152
  - 1.4 常駐型ツールに対抗する 170



2	2 暗号化のテクニック	175
	2.1 暗号化の方法	175
	2.2 復元の方法	193

3	3 プログラムをかくす	195
	3.1 プログラムをVRAM上に置く	195
	3.2 プログラムをスタック上に置く	201
	3.3 メッセージをプログラムに	207

## 3

4	4 錯乱のためのテクニック	213
	4.1 自分自身を転送する	213
	4.2 自分自身にかぶせる	215
	4.3 自分自身を書き換える	220

5	5 ワナをかける	221
	5.1 書き換えを無効にする	221
	5.2 バンク切り替えを使う	225
	5.3 タイマ割り込みを使う	229
	5.4 スタックを書き換える	233

## 5

6	6 既存の知識を破棄させる	237
	6.1 割り込みベクタをすり替える	237
	6.2 パラメータインタフェースを変える	239

7	7 プログラム実行のテクニック	243
	7.1 プログラムを並行実行する	243
	7.2 裏で本物を走らせる	249

## 7

## 資料編

## 1

- 1 CPU 255
  - 1.1 搭載されるCPU 255
  - 1.2 メモリ管理 257
  - 1.3 レジスタ 260
  - 1.4 アドレッシングモード 264
  - 1.5 その他のことから 266

## 2 OS 269

- 2.1 OS起動のメカニズム 269
- 2.2 MS-DOSの構造 273
- 2.3 DISK BASICの構造 280

## 2

## 3

- 3 マシン 283
  - 3.1 割り込み 283
  - 3.2 I/O 284
  - 3.3 その他のことから 284

## 付録

- A SYMDEB機能一覧 290
- B PC-9801 割り込み一覧 292
- C PC-9801 I/O一覧 294
- D INDEX 296

## 基礎編





1. 解説にあたって
2. 解説の定石
3. 解説支援ツール

---

基礎編では、プログラムを解説する立場に立ったプログラム解説に関する基礎的なことがらについて触れます。プログラムの解説が必要な理由や解説の手段、定石など、知っておかなければならないことばかりです。また、解説の際に便利に使える種々のユーティリティも掲載しました。ぜひご活用ください。

# 1

## 解読にあたって

基礎編のはじめでは、解読という行為全般に対して解説します。そもそも解読が必要であるというのはなぜか、解読の方法は決まっているのかなどという疑問があるかと思います。そこで、ここでは解読の必要性と、いくつか考えられる解読の手段について紹介したいと思います。

## 1.1 なぜ解読が必要か

そもそもなぜ解読が必要なのか？ 本書の序章でも書きましたが、ソフトウェアにかけられるプロテクトは、年々（月々？）複雑化の途をたどり、それに対応するコピーのための手段も、次々と高度なものが生み出されています。FDC その他のハードウェアを用いたものには、コピー不能なものがあり、FDC 単体では完全にコピーできないものも存在します。

そこで、コピーツールでは対策として、プロテクトの成否をチェックしている部分（これをチェックルーチンと呼ぶ）を手作業によって捜し出し、この部分を無効にしてしまうという方法が考えられました。このような作業は自動化できないため、あくまでも人間の手によって行われてきたのですが、問題となるのは、チェックルーチンのありかです。チェックルーチンが誰にでも見つけられ、かつ、それが簡単に無効にできるならチェックルーチンの存在意義があり



ません。そこでチェックルーチンを作成するプログラマは、持ち得る技術の限りをつくして、複雑難解なチェックルーチン作りに励むのです。ここでいうチェックルーチンを探す作業を、本書では「解説」と呼んでいます。

さて、プロテクトキラーと呼ばれるプロテクト外し屋は、このチェックルーチンを探して無効とする（これをプロテクトを外す、アンロックにするなどといいます）作業を行うわけですが、チェックルーチンを作成したほうも、あの手この手で引っ掛けてこようとしますから、生半可な気持ちでは外すことができません。そこで両者の闘いが始まるわけです。

もうおわかりでしょう。当然のことのようですが、チェックルーチンの場所というのは決っていません。それどころか形態も決っていません。また、いつ実行されるのかも決っていません。極端な話、その存在すらも解説する側から見たら不明なのです。

そこでチェックルーチンについて知るには、まず解説が必要なのです。チェックルーチンの全貌は、解説した結果、明らかになるといっても過言ではないでしょう。

また、それはチェックルーチンを探すことから始まる高度な知的遊戯ともいえるパズルを楽しめます。みなさんはチェックルーチン作成者が仕掛けるあの手この手のわなをクリアするとき、たまらない面白さを感じることでしょう。

## 1.2 解読にのぞむ

さて、いよいよ解読にのぞむ段階で、いったい何をしたらよいかわからない読者も多いことでしょう。それもそのはず、解読というのは必ずしも必要な作業ではないからです。プログラムは組めても、プログラムを読むことはできない、そのような読者のために、解読の常套手段ともいえることがらについて説明しましょう。

### ■リストをとる

解読と聞けば、まずプログラムの逆アセンブルリストが連想されるでしょう。逆アセンブルリストとは、プログラムを構成する命令の集まりを解釈し、それに対応する命令ニーモニックの列を作成するものです。数値で表現される命令を、私たち生身の人間は理解できませんから（ふつう、理解することはできても瞬時には理解できない）、見てすぐわかるニーモニック（"MOV AX,BX"など）のかたちに変換するわけです。この逆アセンブルリストを作成するには、対象とするソフトウェアの動作形態（動く環境など）を知らなければなりません。ここでは、形態別に逆アセンブルリストのとり方について紹介しましょう。

まず、動作する OS（オペレーティングシステム）のはっきりしている場合です。たとえば、N88-DISK BASIC であるとか、MS-DOS version 3.10 であるとかという場合です。このような場合には、必ず OS に標準で逆アセンブルリストを見るための機能（解読が直接の目的ではないが、とにかく含まれる）が添付されているはずですから、これを使用します。参考までに、DISK

BASIC では MON コマンド内部の L コマンドが、MS-DOS では SYMDEB、あるいは DEBUG コマンド内部の U コマンドを使用することができます。さらに、あくまでも参考までに紹介しておけば、CP/M86 では DDT86 を使用できます。現在、大部分のビジネス用アプリケーションソフト（日本語ワードプロセッサ、データベース、表計算ソフトなど）は、OS をベースとして動作しますので、そのようなプログラムを解読するには、OS 上のツールを使用することができるわけです。

次に、動作する OS がないか、または独自の OS を用意している場合です。このような場合には、IPL と呼ばれるプログラムから解読が必要です（IPL については資料編を参照）。多くのゲームソフトや一部のビジネス用アプリケーションソフトなどが、このようなタイプに属します。とにかく MON コマンドや SYMDEB、DEBUG コマンドのようなものは期待できませんので、他の OS のものを流用するか、専用のツール（市販されている）を使用するしかないでしょう。

OS 上のソフトウェアの逆アセンブルリストをとるには、まずそのプログラムをメモリ上にロードします。ただし、DISK BASIC と MS-DOS とでは手順が異なりますので、別個に説明します。

DISK BASIC の場合、プログラムが BASIC である場合には、MON コマンドを使用する必要はありませんが、機械語プログラムである場合には、それをメモリ上に読み込み、逆アセンブルする必要があります。プログラムの読み込みは、BLOAD コマンドによって行いますが、あらかじめプログラムのロードされるべきセグメントの位置を調べておき、CLEAR, DEF SEG 両コマンドによってプログラム領域を確保しておきます。そして BLOAD コマンドを実行したあと、MON コマンドに入ります。MON コマンド内では L コマンドによってプログラムの逆アセンブルを行います。このと

き、P コマンドによってプリンタ出力が可能な状態にしておけば、画面に表示されるリストが、同時にプリンタへも出力されます。

MON コマンドによってシステムディスク（バージョンによっては含まれていないかもしれない）に含まれる機械語プログラム“mouse.cod”をメモリ上へロードして、逆アセンブルした例を図 1.1 として示します。

■ 図 1.1 “mouse.cod”の逆アセンブル例

```

clear ,&h4000 [ad] ----- 機械語プログラム領域を確保
Ok
def seg=&h4000 [ad] ----- 領域の先頭にセグメントを定義
Ok
bload "mouse.cod [ad] ----- マウスドライバをロード
Ok
mon [ad] ----- モニタに入る
hl 1100 [ad] ----- マウスドライバのエントリを逆アセンブル
0100 E97308      JMP      0976 ----- ジャンプしている
0103 90          NOP
0104 15DF00      ADC      AX,00DF
0107 0000        ADD      [BX+SI],AL
0109 0000        ADD      [BX+SI],AL
010B 0000        ADD      [BX+SI],AL
010D 0000        ADD      [BX+SI],AL
010F 0000        ADD      [BX+SI],AL
0111 0000        ADD      [BX+SI],AL
0113 0000        ADD      [BX+SI],AL
0115 0000        ADD      [BX+SI],AL
0117 0000        ADD      [BX+SI],AL
hl 1976 [ad] ----- ジャンプ先を逆アセンブル
0976 50          PUSH     AX
0977 53          PUSH     BX
0978 51          PUSH     CX
0979 52          PUSH     DX
097A 56          PUSH     SI
097B 57          PUSH     DI
097C 55          PUSH     BP
097D 1E          PUSH     DS
097E 06          PUSH     ES
097F 8CC8        MOV      AX,CS
0981 8EC0        MOV      ES,AX
0983 8B4702      MOV      AX,02[BX]
hl

```

ただし、このとき注意しなければならないことは、MON コマンドにおいては、必ずしも L コマンドが使用できるとは限らないということです。それは、PC-9801U2 以降の機種において、MON コマンドの拡張機能の使用有無がメモリスイッチによって指定できるからで、設定によっては、A,L,E などのコマンドが使用できない

状態になっています。そのような場合には、システムディスクに入っているユーティリティ `switch.n88` を用いて、メモリスイッチの書き換えを行ってください。

また、`L` コマンドでは、逆アセンブル不可能な命令もありますので注意が必要です。逆アセンブルできない命令とは、セグメント外ジャンプ、セグメント外 `CALL`、セグメント外 `RET` 命令です。これらは、リスト上では“?”で表示されますが、続く命令の解釈に食い違いが生じる可能性がありますので、同様に注意が必要です。

MS-DOS の場合では、話は簡単です。SYMDEB あるいは DEBUG を起動する際に、パラメータにロードするプログラム名と、さらにそれに与えるパラメータを並べるのです。そうすれば、SYMDEB/DEBUG のコマンドが表示され、メモリ上にロードされたプログラムが、いつでも実行可能な状態になっていますので、`U` コマンドを用いて逆アセンブルを行うのです。このとき、使用したツールが SYMDEB であれば、`}` コマンドを用いてリストをプリンタやファイルへリダイレクトすることができます。

SYMDEB/DEBUG は、本書では重要な役目を果しますので、その機能について巻末に付録を設けました。参考にしてください。

例として、SYMDEB によるコマンド“`COMMAND.COM`”の逆アセンブル例を、図 1.2 に示しておきます。

以上、簡単に OS 上のプログラムについての、逆アセンブルリストのとり方について説明してきましたが、詳細はそれぞれについてのマニュアルを参照してください。また、逆アセンブルリストと同様に重要なダンプリストも、`MON` コマンド、SYMDEB/DEBUG コマンド双方で、`D` コマンドによってとることができます。プリンタへの出力方法も同様です。

OS を持たないか、独自の OS を持つ場合には、市販されている



■ 図 1.2 "COMMAND.COM"の逆アセンブル例

```

A>symdeb command.com [F] ----- command.com を解析する
Microsoft Symbolic Debug Utility
Version 3.01
(C) Copyright Microsoft Corp 1984, 1985
Processor is [8086]
-u100 [F] ----- COM ファイルはここから逆アセンブル
2F81:0100 E92D0D      JMP     0E30      ----- ジャンプ 命令がある
2F81:0103 BA040B      MOV     DX,0B04
2F81:0106 3D0500      CMP     AX,0005
2F81:0109 741B       JZ      0126
2F81:010B BADC0A      MOV     DX,0ADC
2F81:010E 3D0200      CMP     AX,0002
2F81:0111 7413       JZ      0126
2F81:0113 BAFE0A      MOV     DX,0A7E
-ue30 [F] ----- ジャンプ 先を逆アセンブル
2F81:0E30 BC5D07      MOV     SP,075D
2F81:0E33 B430       MOV     AH,30      ; '0'
2F81:0E35 CD21       INT     21
2F81:0E37 66E0       XCHG    AH,AL
2F81:0E39 3D0A03      CMP     AX,030A
2F81:0E3C 7205       JB      0E43
2F81:0E3E 3D0A03      CMP     AX,030A
2F81:0E41 7612       JBE     0E55
-

```

専用のツールを用いないなら、とりあえずは DISK BASIC の MON コマンドを用いて解析を行ってみましょう。このとき、使用する DISK BASIC のシステムディスクは、できるだけコンパクトなものがよいでしょう（そのほうがメモリを圧迫しないで済むからです）。とりあえずは、必要最小限の機能を備えているものとして、PC-9801M2 用の PC-98H43-MW(K) がよいでしょう。また漢字変換機能などの、必要がないと思われる機能については、メモリ上にロードしないようにします。具体的には、

How many files(0-15)?

の問いに対して、ドライブのふたを開けてリターンキーを押します。

BASIC が起動し MON コマンドを実行したら、とにかく、目的のフロッピディスク上の IPL をメモリに読み込んでみます。

ただしこの場合、必ずしも MON コマンドで読める IPL 形式になっているとは限りません。あらかじめ調べてみる必要があります（IPL の形式については、同様に資料編を参照してください）。

参考までに、IPL の形式を調べる例を紹介します。この例で使用するプログラムは、MON コマンド内部の、A コマンドで入力できる範囲の短いものですので、いちいちセーブする必要もないと思います。プログラムリストと実行手順、IPL 形式の獲得については、図 1.3 のオペレーション例を参照してください。

■ 図 1.3 IPL の形式を得る

```

clear ,&h4000 [a] ----- 機械語領域を確保
Ok
def seg=&h4000 [a] ----- 領域にセグメントを定義
Ok
mon [a] ----- モニタに入る
hla0 [a] ----- 先頭からアSEMBル
0000 31C0          xor ax,ax
0002 8EC0          mov es,ax
0004 26            es:
0005 A08405        mov al,[584]
0008 B40A          mov ah,0a
000A 30C9          xor cl,cl
000C 30F6          xor dh,dh
000E 50            push ax
000F 51            push cx
0010 52            push dx
0011 CD1B          int lb
0013 2E            cs:
0014 C60600100     mov [100],byte 00
0019 2E            cs:
001A 882E0101      mov [101],ch
001E 5A            pop dx
001F 59            pop cx
0020 58            pop ax
0021 730E          jnb 31 ----- エラーでなければ終了
0023 B44A          mov ah,4a
0025 CD1B          int lb
0027 2E            cs:
0028 FE060001      inc byte [100]
002C 2E            cs:
002D 882E0101      mov [101],ch
0031 CC            int 3 ----- 実行を終了
0032
hlg0,31 [a] ----- DISK BASIC (2HD) のディスクで実行してみた
*4000:0031
hld100 [a] ----- 結果をみると単密、セクタ長0となっている
0100 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
hlg0,31 [a] ----- MS-DOS (2HD) のディスクで実行してみた
*4000:0031
hld100 [a] ----- 結果をみると倍密、セクタ長3となっている
0100 01 03 00 00 00 00 00 00 00 00 00 00 00 00 00
hl

```

100H...単密=0  
倍密=1  
101H...セクタ長

IPL の形式を判別したら、その IPL をメモリ上へ読み込みます。ここで読み込む例を示します。参考にしてください (5 インチ 2HD,セクタ長3 の場合)。なお、本来 IPL の読み出される位置は、

資料編で示してあるとおり、1FC0H：0000 あるいは 1FE0H：0000 です。読み込めたら逆アセンブル作業に入り、実行を追跡してください。

## ■リストを読む

逆アセンブルリストがとれたら、まずリストを読んでみます。リストを読むには、当然、頭の中で何が行われているかを確認しながらいくわけですが（頭の中で動作をシミュレートする）、このときいくつかの障害が現れます。これはリストを読むときには常についてまわるものです。障害とは以下にあげるものです。

- ① コードとデータの分離
- ② 動作とアルゴリズムの対比
- ③ 読みにくさ
- ④ マシンの知識
- ⑤ 意図的な妨害

①は、いま読んでいるところが読むべきコード領域なのか、無視すべきデータ領域なのかがわからなくなるということです。これを解決するには、まずプログラムの逆アセンブルを行う前に、ダンプを行ってみるということです。ダンプによってメッセージなどがあれば、その周辺はデータ領域であると推測できます（データが暗号化されている場合、この方法は使えません）。

②は、命令を解説していても、それが何をやっているのかわからなくなる場合があります。「そんなものは当人のスキル不足である」といわれればそれまでですが、経験から流れを掴むしかないようです。とにかく場数をこなせば、カンが冴えてくるはずです。

場数をこなせば、おぼろげながらも「ここは怪しい」、「ここはプロテクトとは無関係だな」などと推測できるようになり、それが的確であれば、大幅な作業の簡略化になるわけです。もっとも確固たる根拠のない、いい加減なものですから濫用は禁物です。

③は、プログラミングを行った人間のスキル不足による、下手なプログラムに出合った場合です。とにかく必要以上に複雑で、かつ適切な方法を用いないなど、理解の及ばない範囲にあるものが該当します。しかし、わざと汚いプログラムにしていたり、また実行効率を優先して美しさを捨てている場合もありますから、いちがいにそうとはいいきれません。

④は、マシンに独自の割り込み命令やハードウェアへの頻繁なアクセスなど、それに対抗するための知識がない場合には、解説が困難になるということです。とにかく、資料を揃えて解説にのぞむか、それなりの知識を頭に叩き込んでおくしかありません。少なくとも何に関連する仕事なのか、ぐらいいはすぐわかるようにする必要があります。

⑤に該当するのは、本書で紹介する意図的にプログラムが読めないようにしたものに出会った場合です。意図的に読めないようにしているのですから、読めるからといって安心はできません。読めることが実は大きな落とし穴だったりするわけです。

以上の障害を意識してリストを読み進むわけですが、人間の処理できる能力を越えたプログラムの場合、解説は停止せざるを得なくなります。長大なプログラムを1から解説するのでは無駄が多すぎます。できれば必要な箇所だけの解説で済ませたいものです。いずれにしても解説は、最初の部分と要所要所のみと考えたほうがよいでしょう。

## ■実行を追う

リストを頭で追うのに対し、こちらはプログラムを実際に実行させながらそのようすを追うというものです。もちろん、ある程度の刻みをもって実行させ、その途中経過（レジスタの変化、実行中の命令など）を追います。この場合、人間が判断すべき分岐命令などは、実際の実行状況に応じて自動的に判断されますから、間違いが起こりにくく、かつ手早く行えるという利点を持ちます。しかし、すべてのプログラムが実行追跡可能なわけではありませんし、場合によっては暴走してしまいます（割り込みを使ったプログラムや、プログラムのロード位置を固定していて、システムを破壊してしまう場合など）。

特にトレースモードによって実行を1命令ごとに追うのは、サブルーチンや割り込み処理ルーチンまでのすべての実行を追うことになってしまいます。正常に復帰できなかったり、とんでもないところまで追跡を行ったりして効率よくありません（SYMDEBではそのような状況を解決するために、サブルーチンや割り込みを1個の命令とみなして実行してしてから、結果を表示するようなコマンドも用意されています）。

また、ブレークポイントを置いて実行を中断しようとしても、実行がブレークポイントに達しなければプログラムは停止せず、どんどん先に進んでしまいます（SYMDEBでは、複数のブレークポイントを置くことで対応）。

以降、基礎編では解説の際の参考になるような知識、テクニックを紹介していきます。話をまとめるために、対象のOSをMS-DOS、解説用ツールをSYMDEB/DEBUGに限定します。しかし、見方を変えればDISK BASICやその他のOSでも応用可能ですから、参考にしてください。





# 2

## 解説の定石

プログラムを解説するにあたっては、いくつか定石ともいえることがありますが。ここでは、この定石について紹介します。

## 2.1 ファイルを調べよう

MS-DOS では、プログラムはファイルのかたちでフロッピーディスク上に存在し、また、MS-DOS にかかわる多くのファイルが、フロッピーディスク上に存在します。ファイルについて調べてみるのが解説の第一歩でしょう。

### ■IO. SYS, MSDOS. SYS, COMMAND. COM

MS-DOS を構成する主なパーツは、IO.SYS、MSDOS.SYS、COMMAND.COM の3つです。これらには手を加えないことが暗黙の規則と決まっているようですから、疑う必要はないでしょうが、徹底するなら一応チェックしておきましょう。見るべき点は、タイムスタンプとサイズです。

さっそく DIR コマンドを用いて、解析しようとするシステムディスクのファイル一覧を見えます。多くの場合、表示は図 2.1 の

ようになるか、あるいは大なり小なり似たものになるでしょう。

■図 2.1 DIR コマンドによる表示

```
A>dir a: 
```

ードライブAのディレクトリー一覧をみる

ドライブ A: のディスクのボリュームラベルはありません。  
ディレクトリは A:\

```

COMMAND COM 24145 86-10-13 0:00
ASSIGN COM 1787 86-10-13 0:00
ATTRIB EXE 8262 86-10-13 0:00
BACKUP EXE 22570 86-10-13 0:00
CHKDSK EXE 9760 86-10-13 0:00
COPY2 COM 3344 86-10-13 0:00
COPYA COM 1319 86-10-13 0:00
CUSTOM COM 5737 86-10-13 0:00
DICM COM 21248 86-10-13 0:00
DISKCOPY COM 6896 86-10-13 0:00
DUMP COM 1999 86-10-13 0:00
EDLIN EXE 7426 86-10-13 0:00
EXE2BIN EXE 2880 86-10-13 0:00
FC EXE 14394 86-10-13 0:00
FIND EXE 6525 86-10-13 0:00
FORMAT EXE 35070 86-10-13 0:00
JOIN EXE 8946 86-10-13 0:00
KEY COM 4591 86-10-13 0:00
LABEL EXE 2918 86-10-13 0:00
MORE COM 319 86-10-13 0:00
PRINT EXE 8472 86-10-13 0:00
RECOVER EXE 4381 86-10-13 0:00
RESTORE EXE 20888 86-10-13 0:00
SPEED COM 1209 86-10-13 0:00
SUBST EXE 9864 86-10-13 0:00
SWITCH COM 2441 86-10-13 0:00
SYS EXE 2917 86-10-13 0:00
RENDIR COM 2668 86-10-13 0:00
USKCGM COM 4181 86-10-13 0:00
SHARE EXE 7904 86-10-13 0:00
SORT EXE 1680 86-10-13 0:00
MSASSIGN COM 1518 86-10-13 0:00
LINK EXE 41114 86-10-13 0:00
SYMDEB EXE 36538 86-10-13 0:00
MAPSYM EXE 51904 86-10-13 0:00
LIB EXE 24138 86-10-13 0:00
MAKE EXE 18675 86-10-13 0:00
NECDIC DRV 31190 86-10-13 0:00
NECDIC SYS 520192 86-10-13 0:00
MENU COM 7092 86-10-13 0:00
MSDOS MNU 1614 86-10-13 0:00
SAMPLE MNU 3548 86-10-13 0:00
RAMDISK SYS 3056 86-10-13 0:00
RSDRV SYS 1797 86-10-13 0:00
MOUSE SYS 2998 86-10-13 0:00
MOUSE DOC 2851 86-10-13 0:00
NECREN DRV 64375 86-10-13 0:00
FILECONV EXE 31648 86-10-13 0:00
README DOC 1531 86-10-13 0:00
CONFIG BAK 81 87-01-27 17:18
AUTOEXEC BAT 12 86-10-13 0:00
CONFIG SYS 82 87-01-27 17:19

```

52 個のファイルがあります。  
60416 バイトが使用可能です。

A&gt;



図 2.1 からわかるように、多くの場合、COMMAND.COM は表示されても IO.SYS と MSDOS.SYS は表示されません。これは、IO.SYS と MSDOS.SYS には、不可視属性というものが施されているからです。不可視属性とは、ファイルを白日の下にさらさないようにするためのもので、ふつうのファイルには付いていません。

しかし、これら 2 つのファイルは特に表面に出す必要もなく、変に目立って削除されてしまっても困るので、このような属性が付いているのです。では、図 2.2 のような操作を行ってください。

■図 2.2 不可視属性を解除する

```

A>symdeb [F5] ----- デバッガを起動
Microsoft Symbolic Debug Utility
Version 3.01
(C) Copyright Microsoft Corp 1984, 1985
Processor is [8086]

-r [F5] ----- フログラムを打ち込む
30B9:0100 mov ax,4301
30B9:0103 mov dx,200 ----- 属性設定を行う簡単なプログラム
30B9:0106 int 21
30B9:0108 int 3
30B9:0109

-e 200 "io.sys" 0 [F5] ----- ファイル名を設定
-g 100 [F5] ----- 実行してみる
AX=FF00 BX=0000 CX=0000 DX=0200 SP=CE36 BP=0000 SI=0000 DI=0000
DS=30B9 ES=30B9 SS=30B9 CS=30B9 IP=0108 NV UP EI PL NZ NA PO NC
30B9:0108 CC INT 3
-e 200 "msdos.sys" 0 [F5] ----- ファイル名を設定
-g 100 [F5] ----- 実行
AX=FF00 BX=0000 CX=0000 DX=0200 SP=CE36 BP=0000 SI=0000 DI=0000
DS=30B9 ES=30B9 SS=30B9 CS=30B9 IP=0108 NV UP EI PL NZ NA PO NC
30B9:0108 CC INT 3
-q [F5] ----- デバッガを脱ける

A>dir a: [F5] ----- もう一度ディレクトリの一覧をみる

ドライブ A: のディスクのボリュームラベルはありません。
ディレクトリは A:\

IO      SYS      32768   86-10-09   18:19 ----- 表示された
MSDOS   SYS      28112   86-10-13   11:37
COMMAND COM     24145   86-10-13   0:00
ASSIGN  COM      1787   86-10-13   0:00
ATTRIB  EXE     8262   86-10-13   0:00
BACKUP  EXE    22570   86-10-13   0:00
CHKDSK  EXE     9760   86-10-13   0:00
COPY2   COM     3344   86-10-13   0:00
COPYA   COM     1319   86-10-13   0:00
CUSTOM  COM     5737   86-10-13   0:00
DICM    COM    21248   86-10-13   0:00
DISKCOPY COM    6896   86-10-13   0:00
DUMP    COM     1999   86-10-13   0:00
EDLIN   EXE     7426   86-10-13   0:00
EXE2BIN EXE     2880   86-10-13   0:00

```

---

```

FC           EXE      14394  86-10-13  0:00
FIND         EXE      6525  86-10-13  0:00
FORMAT       EXE     35070  86-10-13  0:00
JOIN         EXE      8946  86-10-13  0:00
KEY          COM      4591  86-10-13  0:00
LABEL        EXE      2918  86-10-13  0:00
MORE         COM       319  86-10-13  0:00
PRINT        EXE     8472  86-10-13  0:00
RECOVER      EXE     4381  86-10-13  0:00
RESTORE      EXE    20888  86-10-13  0:00
SPEED        COM     1209  86-10-13  0:00
SUBST        EXE     9864  86-10-13  0:00
SWITCH       COM     2441  86-10-13  0:00
SYS          EXE     2917  86-10-13  0:00
RENDIR       COM     2668  86-10-13  0:00
USKCGM       COM     4181  86-10-13  0:00
SHARE        EXE     7904  86-10-13  0:00
SORT         EXE     1680  86-10-13  0:00
MSASSIGN     COM     1518  86-10-13  0:00
LINK         EXE    41114  86-10-13  0:00
SYMDEB       EXE    36538  86-10-13  0:00
MAPSYM       EXE    51904  86-10-13  0:00
LIB          EXE    24138  86-10-13  0:00
MAKE         EXE    18675  86-10-13  0:00
NECDIC       DRV    31190  86-10-13  0:00
NECDIC       SYS    520192 86-10-13  0:00
MENU         COM     7092  86-10-13  0:00
MSDOS        MNU     1614  86-10-13  0:00
SAMPLE       MNU    3548  86-10-13  0:00
RAMDISK      SYS    3056  86-10-13  0:00
RSDRV        SYS    1797  86-10-13  0:00
MOUSE        SYS    2998  86-10-13  0:00
MOUSE        DOC    2851  86-10-13  0:00
NECREN       DRV    64375  86-10-13  0:00
FILECONV     EXE    31648  86-10-13  0:00
README       DOC     1531  86-10-13  0:00
CONFIG       BAK       81  87-01-27 17:18
AUTOEXEC     BAT       12  86-10-13  0:00
CONFIG       SYS       82  87-01-27 17:19

```

54 個のファイルがあります。  
60416 バイトが使用可能です。

A>

---

ここでもう一度、DIR コマンドを実行してください。一覧に 2 つのファイルが表示されたはずです。表示されたのが確認できたら、タイムスタンプとサイズをメモしておき、オリジナルの MS-DOS のものと比較します。異なっていれば要チェック、同じであっても要チェックです。たとえタイムスタンプとサイズが同じでも、これらはどうにでも操作できます。油断はできません。

また、たとえタイムスタンプやサイズが異なっていても、それは MS-DOS 自体のバージョンが異なる、ということを意味している可能性もありますので、あたまから疑ってかかるのも考えものです。

タイムスタンプとサイズが同じ場合には、FC コマンドを用いてファイル相互の比較を行ってみましょう。図 2.3 の例は『一太郎 Ver2』『松 86』のシステムディスクと、オリジナルの MS-DOS のシステムディスクを比較したものです。

■図 2.3 ファイルの比較

```

A>fc io.sys b:io.sys [a] -----オリジナルのID.SYSと一太郎のID.SYSを比較
fc: 違いは見つかりません。 -----手は加えられていない

A>fc msdos.sys b:msdos.sys [a] -----MSDOS.SYSについても同様に調べる
fc: 違いは見つかりません。 -----これにも手は加えられていない

A>fc io.sys b:io.sys [a] -----今度は松86で比較
00004CC0: E9 D1
00004CC1: 67 E0
00004CC2: 0E 03
00004CC3: 90 C8
00004D21: E9 D1
00004D22: 10 E0
00004D23: 0E 03
00004D24: 90 C8
00004D30: E9 83
00004D31: 0B C1
00004D32: 0E 04
0000582A: 21 00
0000582B: E0 00
0000582C: 01 00
0000582D: C1 00
0000582E: 83 00
0000582F: D2 00
00005831: E9 00
00005832: 90 00
00005833: F1 00
00005834: D1 00
00005835: E0 00
00005836: 01 00
00005837: C1 00
00005838: 83 00
00005839: D2 00
0000583B: E9 00
0000583C: E7 00
0000583D: F1 00
0000583E: 83 00
0000583F: C1 00
00005840: 04 00
00005841: 83 00
00005842: D2 00
00005844: E9 00
00005845: EC 00
00005846: F1 00

-----大量のバッチ!

A>fc msdos.sys b:msdos.sys [a]
fc: 違いは見つかりません。 -----MSDOS.SYSについては大丈夫であった

A>

```

## ■CONFIG.SYS

CONFIG.SYS ファイルは、デバイスドライバの登録を行ったり、コマンドプロセッサの指定を行うためのファイルで、システム再構築ファイルと呼ばれます。実際、上の3つのファイルよりも、このファイルを優先して調べたほうがよいでしょう。注目するのは、

DEVICE

SHELL

のいずれかで始まる行です。DEVICE は、デバイスドライバを指定するための行であり、SHELL はコマンドプロセッサを指定するための行です。特に注意するのは DEVICE 行であり、ここに何か変わったファイルが登録してあればチェックしておきましょう。

『一太郎 Ver2』の ATOK5A.SYS, ATOK5B.SYS などのように、マニュアルで公開され、その用途が明白であるものならばよいのですが、そうでなければ要チェックです。

そもそも SHELL 行はない場合が多く、たとえあったとしても、COMMAND.COM がそのまま指定されている場合が多いので、特に気にする必要はないかも知れません。

## ■AUTOEXEC.BAT

AUTOEXEC.BAT を調べることは、最初に行われるプログラムを調べることになります。たいていのアプリケーションでは、このファイルによりアプリケーションの起動が自動化されています。

たとえば、『松 86』の AUTOEXEC.BAT ファイルの内容は、次のようになっています。

MATU

これは、何の細工もなしに MATU.COM を起動させることを指示しています。ふつうはこんなものです。

## ■かくされたファイル

MS-DOS のディスクには、IO.SYS などのように隠されたファイルが多数存在する可能性があります。ちなみに『1-2-3 リリース 2J』などもこのようなファイルが多数存在し、プロテクトの面で大きな活躍をしています。

このようなファイルは DIR コマンドでは表示されませんので、専用のプログラムが必要となります。ここで、カレントドライブのカレントディレクトリ中にある不可視属性の付いたファイルの不可視属性を、すべて解除してしまうユーティリティ DIG.COM を、図 2.4 として紹介します。実行は、コマンド名のみで OK です。

■図 2.4 DIG.ASM ソースリスト

```

;
;*****
;
;      DIG.ASM
;
;      カレントドライブ、カレントディレクトリ内のファイルを正規化
;
;      COPYRIGHT(C) 1987 BY SHUWA SYSTEM TRADING CO.,LTD.
;
;      LAST MODIFIED ON FEBRUARY 2ND,1987
;*****
;
CODE    SEGMENT
        ASSUME    CS:CODE,DS:CODE,ES:CODE,SS:CODE
;
;      ORG        100H
;
DIG     PROC
;
;      MOV        AH,4EH          ; 最初のファイルを検索
;      MOV        CX,37H          ; ボリュームラベルを除く属性を対象とする
;      LEA        DX,DEFAULT_PATH ; 検索対象のパス名
;      INT        21H
;      JC         DIG_EXIT        ; ファイルが見つからない場合
;
;      CALL       RESET_ATTR      ; 属性を解除

```

```

;
DIG_LOOP:
MOV     AH,4FH           ; 続くファイル名を検索
INT     21H
JC      DIG_EXIT         ; ファイルがもうない場合
;
CALL    RESET_ATTR      ; 属性を解除
JMP     DIG_LOOP         ; 次のファイルへ
;
DIG_EXIT:
MOV     AX,4C00H         ; 非常駐終了
INT     21H
;
RESET_ATTR PROC NEAR    ; システム、不可視、読み出し専用属性を解除
MOV     BX,80H           ; デフォルトのDTAのアドレス
MOV     AX,4301H         ; 属性変更
MOV     CX,[BX+15H]       ; もとの属性
AND     CX,0F8H           ; 必要な属性を残して解除
LEA     DX,[BX+1EH]       ; 解除するファイル名
INT     21H
RET
RESET_ATTR ENDP
;
DEFAULT_PATH DB ' *.*',0 ; 検索対象のファイル名
;
DIG      ENDP
;
CODE    ENDS
;
END      DIG

```

■図 2.4 DIG.COM ダンプリスト

```

00000000 : B4 4E B9 37 00 8D 16 33 01 CD 21 72 0E E8 10 00 : 52F
00000010 : B4 4F CD 21 72 05 E8 07 00 EB F5 B8 00 4C CD 21 : 729
00000020 : BB 80 00 B8 01 43 8B 4F 15 81 E1 F8 00 8D 57 1E : 682
00000030 : CD 21 C3 2A 2E 2A 00 : 233

```

また、マニュアルには公開されていませんが、サブディレクトリもこの属性が付きます。ディレクトリ一覧をとって見て、ファイルの量やサイズの総計と残り容量が不釣り合いな場合には要注意です。プログラム DIG は、サブディレクトリに付いている不可視属性も取り去るので安心です。なお、サブディレクトリに対する属性は、システムコールで設定することができません。SYMDEB/DEBUGなどを用いて、直接に属性の書き換えを行わなければならないようです。

## ■異常なファイル

そもそもコピーが禁止されているソフトウェアでは、すべてのファイルがコピー可能であるとは限りません。そこで、すべてのファイルをダンプするなりして正常に読み出せることを、チェックするようお勧めします。なお、念のためにいえば、確実にデータの記録されている領域をアクセスしないと意味がありません。

ディスクの異常ではなく正常に読み出せないファイルがあれば、何らかの面でプロテクトに利用されていると思ってよいでしょう。

## 2.2 まずは比較検討を

ファイルをひとつおりチェックしたら、適当なコピーツールでバックアップを試みてみましょう。もっとも、皆さんは解読しようなどと思うぐらいですから、すでにバックアップは試みてあって、そのへんに1枚くらいはころがっているかもしれません。さて、まずはオリジナルを実行してください。正常に動作します。

続けてバックアップを実行してください。正常に動作しませんね。その違いや経過をメモしておいてください。暴走したら暴走、エラーメッセージを出力してMS-DOSへ戻ったならエラーと記録しておくのです。このようなはっきりした兆候はもちろん、ふつうに操作したときのようすも記録しておきましょう。暴走もせずエラーも出さないからといって安心はできません。

皆さんはまったく同じ操作を行って、まったく同じように動作するかどうかを確認します。

一見無駄なようなこの観察が、あとで必ず役に立ちます。

## 2.3 INT 1BH発見がポイント

INT 1BHというのは、あまりにも有名なディスクアクセスを行うための割り込み命令です。PC-9801では、この命令はディスク BIOS に割り当てられており、この命令を実行することによって、ディスク BIOS を使用することができるわけです（詳細は既刊『PC-9800 シリーズ ザ・プロテクト』を参照）。

なぜ INT 1BH が重要かということ、プロテクトのチェックに、この INT 1BH を用いている場合が多いからです。INT 1BH はかなり細かな操作まで行うことができるため、チェック程度であれば INT 1BH で十分事足ります。この INT 1BH を探すことは、プロテクトチェックを行っている場所を探すことになります。

INT 1BH は容易に探すことができます。プロテクトチェックを行っていきそうなプログラムファイルをデバッガで読み込み、デバッガの検索機能を使って位置を見つけます。例として、FORMAT.EXE 内の INT 1BH を搜してみます（FORMAT.EXE にプロテクトチェック機能が含まれているわけではありませんが）。図 2.5 に示すオペレーションは、FORMAT.EXE 内の INT 1BH を搜してみた例です。

基本的にチェックルーチン探しは、INT 1BH 探しから始めればよいのですが、これでも十分ではありません。それは、あらかじめ INT 1BH が搜されるであろうことを想定したチェックルーチンが、ふつうとなっているからです（とはいえ多くのソフトウェアはこの段階でチェックルーチンを外されてしまいます）。これについては応用編で触れます。



## ■図 2.5 INT 1BH を探す

```

A>symdeb format.exe [F5] ----- コマンドファイルを指定して読み込む
Microsoft Symbolic Debug Utility
Version 3.01
(C) Copyright Microsoft Corp 1984, 1985
Processor is [8086]
-r [F5] ----- プログラムのサイズを確かめる ----- 82F6H バイトだ
AX=0000 BX=0000 CX=82F6 DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=30C9 ES=30C9 SS=3100 CS=3110 IP=0000 NV UP EI PL NZ NA PO NC
3110:0000 BC0001 MOV SP,0100
-s0 82f6 od 1b [F5] ----- プログラムサイズだけ "CD" "1B" という並びを探す

30C9:06DA
30C9:071C
30C9:0741
30C9:187F
30C9:188B
30C9:1AFA
30C9:1BAD
30C9:1BC1
30C9:1BDF
30C9:1C10
30C9:1C24
30C9:1C3F
30C9:1C56
30C9:1CB5
30C9:1CF2
30C9:1D06
30C9:1DA4
30C9:1DB5
30C9:1DC4
30C9:1DCB
30C9:1DD9
30C9:1E1F
30C9:1E2F
30C9:1F07
30C9:1FBD
30C9:1FD1
30C9:288C
30C9:2899
30C9:29E2
30C9:29FC
30C9:2B54
30C9:2B60
30C9:2BCF
30C9:2BCF
30C9:43AA
30C9:498F
30C9:4B43
30C9:4B96
30C9:647C
30C9:6489
30C9:65ED
30C9:660F
30C9:66FA
30C9:6706
30C9:6815
30C9:6821
30C9:6EDD
30C9:6EE9
30C9:6F15
30C9:6F21
30C9:6F3B
30C9:71AC
30C9:71B8
30C9:788E
30C9:7C5D

```

これだけあるノ

## 2.4 環境変数を調べる

環境変数とは、コマンドの動作を外部から指定するためのもので、`PATH` や `PROMPT` がその代表格です。環境変数の働きを理解するには、`PATH` や `PROMPT` などの働きを理解することが最も近道です。`PATH` は、コマンドを検索するパスを保持するもので、`PROMPT` はプロンプト（`"A>"` など）の形式を保持するためのものです。すなわち、情報をすぐには変更しないが、できるだけ柔軟に状況に対応できる方法はないか、というのが環境なのです。

環境は、`PATH` や `PROMPT` などの標準的なもののほか、`LIB` や `INCLUDE` などのように、特定のアプリケーションに参照が限定されるものもあります。`LIB` は `LINK.EXE` などと参照され、ライブラリファイルの存在するパスを表しますし、`INCLUDE` は C 言語のコンパイラで参照され、ヘッダファイルの存在するパスを表します。いちど環境変数に設定しておけば、コマンド起動時に明示的に指定してやる必要はないのです。

解読を行うには、この環境変数をチェックすることも必要です。なぜなら、あるアプリケーションにのみ有効な環境変数が設定されていなければ、動作しないことも考えられるからで、アプリケーションを正常に動作させた際、環境変数のようすを確認してみることも必要です。環境変数は、`SET` コマンドで見ることができます。

# 3

## 解説支援ツール

基礎編の最後は、プログラム解説に便利なユーティリティ群を取り上げましょう。これらはチェックルーチン捜しだけでなく、通常のプログラムの解説にも十分役立つはずです。

## 3.1 実行中断・レジスタ値表示

キー入力割り込みを利用して、任意の位置におけるレジスタの内容を表示するユーティリティ DREG.COM を紹介します。これは、PC-9801 に存在する COPY キーの機能を、COPY キーを押した時点でのレジスタ値、またはメモリ値の内容をプリンタへ打ち出すものです。

プログラムの動作原理は単純です。COPY キーが押されると、“INT 06H”に対応する割り込みが発生することを利用して、対応する割り込みベクタを DREG の用意するルーチンのアドレスへ書き換えておきます。それ以降で COPY キーが押されれば、DREG 内のルーチンへ制御が移りますから、そこで、レジスタ値のプリンタ出力などの処理を行えばよいわけです。レジスタ CS、レジスタ IP の内容の取り出しについては、3.2 を参照してください。

DREG の詳しい使用法を説明します。まず、何もパラメータを与えずに DREG を起動します。すると、COPY キーを押したときに出力したいメモリの位置を聞いてきますので、アドレスとサイズ

を次の形式で入力してください。

XXXX : YYYY, B/W/D

ここで XXXX はセグメントベース、YYYY はオフセットアドレスです。また、”,”のあとには B か W か D を指定してください。B を指定した場合には 1 バイト (ZZ)、W を指定した場合には 2 バイト (ZZZZ)、D を指定した場合には 4 バイト (ZZZZ : ZZZZ) のメモリ内容を出力します。これは連続して聞いてきますので、終わるときにはリターンキーのみを入力してください。これらの指定は 10 個まで行うことができます。

ただし、DREG には使用上の制限があり、DREG を実行した後に、COPY キー割り込みに対応した割り込みベクタを書き換えられれば、当然機能は失われます。また、プリンタ出力中に COPY キーを押せば、プリンタへの出力が不安定になることがあります。

ために、DREG を DISKCOPY.COM の実行中に呼び出してみました。そのようすを図 3.1 として示しておきます。

■図 3.1 DISKCOPY.COM の実行を追う

```

A>DREG [F5] ----- DREG を常駐させる

アドレスを入力して下さい (SEGMENT:OFFSET,B/W/D): 9000:0000,B
アドレスを入力して下さい (SEGMENT:OFFSET,B/W/D): 9000:0000,W
アドレスを入力して下さい (SEGMENT:OFFSET,B/W/D): 9000:0000,D
COPYキーの機能を変更しました。
-----
A>DISKCOPY A: B: [F5] ----- DISKCOPY を動かす。CS に注目

AX=0001 BX=0018 CX=92F5 DX=2890 SI=0564 DI=1AB4 BP=0838 SP=082E IP=2192
CS=FD80 DS=0000 ES=FD80 SS=0664 FLAGS=F246
MEMORY: 9000:0000 00
MEMORY: 9000:0000 0000
MEMORY: 9000:0000 0000:0000
-----
AX=0002 BX=0018 CX=3C31 DX=2890 SI=056C DI=1AB4 BP=0838 SP=082C IP=2190
CS=FD80 DS=0000 ES=FD80 SS=0664 FLAGS=F246
MEMORY: 9000:0000 3F
MEMORY: 9000:0000 413F
MEMORY: 9000:0000 A232:413F

```

一定アドレスを持った形式で出力してみる

コピー元のディスクから読み始めたところで出力メモリは0のままである

読み込みが終わり、コピー先に書き始めたところで出力メモリ内にデータが確認できる

```

AX=0002 BX=0018 CX=778F DX=2890 SI=056C DI=1AB4 BP=0830 SP=0824 IP=2190
CS=FD80 DS=0000 ES=FD80 SS=0664 FLAGS=F246
MEMORY: 9000:0000 8B
MEMORY: 9000:0000 468B
MEMORY: 9000:0000 0B19:468B |-----2回目の読み込み、データが変化している

AX=0002 BX=0018 CX=2C1F DX=2890 SI=056C DI=1AB4 BP=0838 SP=082C IP=2190
CS=FD80 DS=0000 ES=FD80 SS=0664 FLAGS=F246
MEMORY: 9000:0000 8B
MEMORY: 9000:0000 468B
MEMORY: 9000:0000 0B19:468B |-----3回目の読み込みまではここまでなかった
N
A>
AX=0300 BX=033A CX=0001 DX=007F SI=90A3 DI=035E BP=0001 SP=0518 IP=00B9
CS=0D4E DS=0D4E ES=0664 SS=0D4E FLAGS=F206
MEMORY: 9000:0000 8B
MEMORY: 9000:0000 468B
MEMORY: 9000:0000 0B19:468B |-----コマンド待ちのとき

```

### ■図 3.2 DREG.ASM ソースリスト

```

;*****
;
;      DREG.ASM
;
;      COPYキーによる実行の中断・レジスタ値・メモリ値の表示
;
;      COPYRIGHT(C) 1987 BY SHUWA SYSTEM TRADING CO.,LTD.
;
;      LAST MODIFIED ON JANUARY 30TH,1987
;*****
;
MEM_PACK      STRUC      ; アドレスセーブ領域の構造
SIZES         DB         ? ; 表示サイズ
OFFSETS       DW         ? ; 表示アドレスオフセット
SEGMENTS      DW         ? ; 表示アドレスセグメント
MEM_PACK      ENDS
;
COPY_VECT     EQU        5  ; COPYキー割り込みのベクタ番号
MAX_SAVE      EQU        10 ; 最大アドレスセーブ数
;
CODE          SEGMENT
ASSUME        CS:CODE,DS:CODE,ES:CODE,SS:CODE
;
;      ORG        100H
;
DREG          PROC        NEAR
LEA           BX,MEM_INFO ; メモリ情報ブロックの先頭
XOR           SI,SI       ; メモリ情報ブロックのオフセット
XOR           CX,CX       ; メモリ情報ブロックのカウンタ
;
INPUT_ADDRESS:
CMP           CX,MAX_SAVE ; 最大アドレス数を越えたか?
JE            INPUT_END   ; 越えたら入力を終了
;
LEA           DX,PROMPT   ; 入力を促すプロンプトのアドレス
MOV           AH,9
INT           21H
;
LEA           DX,IN_BUFFER ; キーボードから入力

```

```

MOV     AH,10
INT     21H

;
CMP     IN_BUFFER+1,0      ; リターンキーのみの入力が見る
JE      INPUT_END         ; リターンキーのみの入力であった

;
LEA     DI,IN_BUFFER+2    ; 入力バッファの文字部のアドレス
CALL    GET_HEX           ; セグメントアドレスを取り出す
MOV     [BX+SI],SEGMENTS,AX ; セグメントアドレスをセット
MOV     AL,[DI]           ; 区切りがあるか見る
CMP     AL,'.'            ; 区切りは正しいか?
JE      GOOD_DELIMIT      ; 区切りは正常であった

;
DISP_BAD_MESSAGE:
LEA     DX,BAD_INPUT      ; 入力不正であるというメッセージを表示
MOV     AH,9
INT     21H
JMP     INPUT_ADDRESS     ; 入力をもういちどやり直す

;
GOOD_DELIMIT:
INC     DI                ; オフセットアドレスを取り出す
CALL    GET_HEX           ; セグメントアドレスを取り出す
MOV     [BX+SI],OFFSETS,AX ; オフセットアドレスをセット
MOV     AL,[DI]           ; 区切りがあるか見る
CMP     AL,'.'            ; 区切りは正しいか?
JNE     DISP_BAD_MESSAGE ; 区切りが正しくない
INC     DI                ; サイズ指定を取り出す
MOV     AL,[DI]           ; 1文字取り出す
CMP     AL,'a'            ; 英大文字か判断する
JB      ANALYSE_CHAR      ; 英大文字か数字

;
CMP     AL,'z'            ; 英大文字か判断する
JA      ANALYSE_CHAR      ; 英大文字でない

;
SUB     AL,20H            ; 英大文字へ変換

;
ANALYSE_CHAR:
CMP     AL,'B'            ; 1バイト表示の指示か?
MOV     DL,1              ; サイズを1バイトへ
JE      SET_SIZE          ; 1バイト表示の指示

;
CMP     AL,'W'            ; 2バイト表示の指示か?
MOV     DL,2              ; サイズを2バイトへ
JE      SET_SIZE          ; 2バイト表示の指示

;
CMP     AL,'D'            ; 4バイト表示の指示か?
MOV     DL,4              ; サイズを4バイトへ
JNE     DISP_BAD_MESSAGE ; 文字が不正

;
SET_SIZE:
MOV     [BX+SI].SIZES,DL ; サイズをセット
ADD     SI,SIZE MEM_PACK ; 次のアドレスへ
INC     CX                ; アドレス数を増す
JMP     INPUT_ADDRESS

;
INPUT_END:
MOV     AH,25H            ; 割り込みベクタ書き換え
MOV     AL,COPY_VECT
LEA     DX,ENTRY          ; 割り込み処理ルーチンのアドレス
INT     21H
MOV     AH,9              ; 変更した旨のメッセージを表示
LEA     DX,MESSAGE
INT     21H
MOV     AX,3100H          ; 常駐終了
MOV     DX,100H
INT     21H

```

```

;
MEM_INFO      MEM_PACK      MAX_SAVE DUP(<0,0,0>)
;
IN_BUFFER      DB      16      ; キー入力バッファ (最大文字数)
               DB      ?      ; 実際に入力された文字数
               DB      16 DUP(?) ; 文字部
;
PROMPT         DB      13,10
               DB      'アドレスを入力して下さい'
               DB      '(SEGMENT:OFFSET,B/W/D): '
               DB      '$'
;
BAD_INPUT      DB      13,10
               DB      '入力が正しくありません!!'
               DB      7,'$'
;
MESSAGE        DB      13,10
               DB      'COPYキーの機能を変更しました。'
               DB      13,10,'$'
;
SIZE_BUFFER1   =      OFFSET OUT_BUFFER2-OFFSET OUT_BUFFER1
OUT_BUFFER1    DB      13,10,'AX=' ; レジスタ値出力用バッファ
OUT_AX         DB      '0000 BX='
OUT_BX         DB      '0000 CX='
OUT_CX         DB      '0000 DX='
OUT_DX         DB      '0000 SI='
OUT_SI         DB      '0000 DI='
OUT_DI         DB      '0000 BP='
OUT_BP         DB      '0000 SP='
OUT_SP         DB      '0000 IP='
OUT_IP         DB      '0000',13,10
;
SIZE_BUFFER2   =      OFFSET OUT_BUFFER3-OFFSET OUT_BUFFER2
OUT_BUFFER2    DB      'CS=' ; セグメントレジスタ値出力用バッファ
OUT_CS         DB      '0000 DS='
OUT_DS         DB      '0000 ES='
OUT_ES         DB      '0000 SS='
OUT_SS         DB      '0000 FLAGS='
OUT_FLAGS      DB      '0000',13,10
;
SIZE_BUFFER3   =      OFFSET AXSAVE-OFFSET OUT_BUFFER3
OUT_BUFFER3    DB      'MEMORY: ' ; メモリ値出力用バッファ
OUT_SEGMENT    DB      '0000:'
OUT_OFFSET     DB      '0000 '
OUT_MEMORY     DB      9 DUP(?)
               DB      13,10
;
AXSAVE         DW      ?      ; レジスタAXをセーブする領域
BXSAVE         DW      ?      ; レジスタBXをセーブする領域
CXSAVE         DW      ?      ; レジスタCXをセーブする領域
DXSAVE         DW      ?      ; レジスタDXをセーブする領域
SISAVE         DW      ?      ; レジスタSIをセーブする領域
DISAVE         DW      ?      ; レジスタDIをセーブする領域
BPSAVE         DW      ?      ; レジスタBPをセーブする領域
SPSAVE         DW      ?      ; レジスタSPをセーブする領域
DSSAVE         DW      ?      ; レジスタDSをセーブする領域
ESSAVE         DW      ?      ; レジスタESをセーブする領域
SSSAVE         DW      ?      ; レジスタSSをセーブする領域
;
ENTRY PROC FAR ; COPYキーが押下時の割込処理ルーチン
STI ; 割り込みを禁止する
MOV CS:AXSAVE,AX ; レジスタAXをセーブ
MOV CS:BXSAVE,BX ; レジスタBXをセーブ
MOV CS:CXSAVE,CX ; レジスタCXをセーブ
MOV CS:DXSAVE,DX ; レジスタDXをセーブ
MOV CS:SISAVE,SI ; レジスタSIをセーブ

```

```

MOV     CS,DISAVE,DI      ; レジスタDIをセーブ
MOV     CS,BPSAVE,BP      ; レジスタBPをセーブ
MOV     CS,SPSAVE,SP      ; レジスタSPをセーブ
MOV     CS,DSSAVE,DS      ; レジスタDSをセーブ
MOV     CS,ESSAVE,ES      ; レジスタESをセーブ
MOV     CS,SSSAVE,SS      ; レジスタSSをセーブ
;
MOV     BP,SP              ; スタックトップの値をBPへコピー
LEA     BP,[BP+6]          ; 表示不用な部分をスキップする
;
MOV     AX,CS:AXSAVE      ; レジスタの内容を表示(レジスタAX)
LEA     DI,OUT_AX
CALL    SET_HEX
;
MOV     AX,CS:BXSAVE      ; レジスタBXの内容をバッファへセット
LEA     DI,OUT_BX
CALL    SET_HEX
;
MOV     AX,CS:CXSAVE      ; レジスタCXの内容をバッファへセット
LEA     DI,OUT_CX
CALL    SET_HEX
;
MOV     AX,CS:DXSAVE      ; レジスタDXの内容をバッファへセット
LEA     DI,OUT_DX
CALL    SET_HEX
;
MOV     AX,CS:SI SAVE     ; レジスタSIの内容をバッファへセット
LEA     DI,OUT_SI
CALL    SET_HEX
;
MOV     AX,CS:DI SAVE     ; レジスタDIの内容をバッファへセット
LEA     DI,OUT_DI
CALL    SET_HEX
;
MOV     AX,CS:BP SAVE     ; レジスタBPの内容をバッファへセット
LEA     DI,OUT_BP
CALL    SET_HEX
;
MOV     AX,CS:SPSAVE      ; レジスタSPの内容をバッファへセット
ADD     AX,6               ; 割り込み発生に合わせて補正
LEA     DI,OUT_SP
CALL    SET_HEX
;
MOV     AX,[BP]           ; レジスタIPの内容をバッファへセット
LEA     DI,OUT_IP
CALL    SET_HEX
;
LEA     BX,OUT_BUFFER1    ; 汎用レジスタの値をプリンタへ出力
MOV     CX,SIZE_BUFFER1
CALL    OUT_PRINTER
;
MOV     AX,CS:DSSAVE      ; レジスタDSの内容をバッファへセット
LEA     DI,OUT_DS
CALL    SET_HEX
;
MOV     AX,CS:ESSAVE      ; レジスタESの内容をバッファへセット
LEA     DI,OUT_ES
CALL    SET_HEX
;
MOV     AX,CS:SSSAVE      ; レジスタSSの内容をバッファへセット
LEA     DI,OUT_SS
CALL    SET_HEX
;
MOV     AX,[BP+2]         ; レジスタCSの内容をバッファへセット
LEA     DI,OUT_CS
CALL    SET_HEX
;

```



```

MOV     AX,[BP+4]           ; フラグレジスタの内容をバッファへセット
LEA     DI,OUT_FLAGS
CALL    SET_HEX
;
LEA     BX,OUT_BUFFER2     ; セグメントレジスタの値をプリンタへ出力
MOV     CX,SIZE_BUFFER2
CALL    OUT_PRINTER
;
LEA     BX,MEM_INFO        ; アドレス情報領域のアドレス
XOR     SI,SI              ; アドレス情報領域のオフセット
XOR     CX,CX              ; アドレス数をカウンタ
;
OUT_LOOP:
CMP     CX,MAX_SAVE        ; すべてのアドレスを出力したか?
JNE     OUT_LOOP1
JMP     ENTRY_EXIT        ; 出力したら終了
;
OUT_LOOP1:
CMP     CS:[BX+SI].SIZES,0  ; すべてのアドレスを出力したか?
JNE     OUT_LOOP2
JMP     ENTRY_EXIT        ; 出力したら終了
;
OUT_LOOP2:
PUSH    CX
MOV     AX,CS:[BX+SI].SEGMENTS ; セグメントをバッファへセット
LEA     DI,OUT_SEGMENT
CALL    SET_HEX
;
MOV     AX,CS:[BX+SI].OFFSETS ; オフセットをバッファへセット
LEA     DI,OUT_OFFSET
CALL    SET_HEX
;
PUSH    BX
MOV     AL,CS:[BX+SI].SIZES   ; サイズに合わせて出力形式を変える
LES     BX,CS:DWORD PTR [BX+SI].OFFSETS ; 出力対象アドレスを得る
DEC     AL                   ; 1 バイトの出力か?
JZ      OUT_BYTES           ; 1 バイトの出力を行う
;
DEC     AL                   ; 2 バイトの出力か?
JZ      OUT_WORDS           ; 1 ワードの出力を行う
;
MOV     AX,ES:[BX]           ; オフセット部を取り出す (前半)
LEA     DI,OUT_MEMORY+5
CALL    SET_HEX
MOV     CS:OUT_MEMORY+4,':' ; セパレータをバッファへセット
MOV     AX,ES:[BX+2]         ; セグメント部を取り出す (後半)
LEA     DI,OUT_MEMORY
CALL    SET_HEX
JMP     OUT_BUFFERS
;
OUT_BYTES:
MOV     AX,ES:[BX]           ; 内容を取り出す
LEA     DI,OUT_MEMORY
CALL    SET_HEX2
MOV     AX,2020H             ; 残りをスペースで埋める
MOV     CS:WORD PTR OUT_MEMORY+2,AX
MOV     CS:WORD PTR OUT_MEMORY+4,AX
MOV     CS:WORD PTR OUT_MEMORY+6,AX
MOV     CS:OUT_MEMORY+8,AL
JMP     OUT_BUFFERS
;
OUT_WORDS:
MOV     AX,ES:[BX]           ; 内容を取り出す
LEA     DI,OUT_MEMORY
CALL    SET_HEX
MOV     AX,2020H             ; 残りをスペースで埋める

```

```

MOV     CS:WORD PTR OUT_MEMORY+4,AX
MOV     CS:WORD PTR OUT_MEMORY+6,AX
MOV     CS:OUT_MEMORY+8,AL

;
OUT_BUFFERS:                                ; メモリ内容バッファの内容を出力
    LEA     BX,OUT_BUFFER3
    MOV     CX,SIZE_BUFFER3
    CALL    OUT_PRINTER
    ADD     SI,SIZE_MEM_PACK                ; 次のアドレスへ
    INC     CX
    POP     BX
    JMP     OUT_LOOP                        ; 最大アドレス数まで繰り返し

;
ENTRY_EXIT:
    MOV     AX,CS:AXSAVE                    ; レジスタAXを復帰
    MOV     BX,CS:BXSAVE                    ; レジスタBXを復帰
    MOV     CX,CS:CXSAVE                    ; レジスタCXを復帰
    MOV     DX,CS:DXSAVE                    ; レジスタDXを復帰
    MOV     SI,CS:SISAVE                    ; レジスタSIを復帰
    MOV     DI,CS:DISAVE                    ; レジスタDIを復帰
    MOV     BP,CS:BPSAVE                    ; レジスタBPを復帰
    MOV     DS,CS:DSSAVE                    ; レジスタDSを復帰
    MOV     ES,CS:ESSAVE                    ; レジスタESを復帰
    MOV     SS,CS:SSSAVE                    ; レジスタSSを復帰
    MOV     SP,CS:SPSAVE                    ; レジスタSPを復帰

;
    CLI                                         ; 割り込みを許可
    IRET
ENTRY     ENDP

;
OUT_PRINTER PROC NEAR                        ; プリンタへの出力
    PUSH    ES
    MOV     AX,CS
    MOV     ES,AX
    MOV     AH,30H                          ; プリンタへのブロック出力
    INT     1AH
    POP     ES
    RET
OUT_PRINTER ENDP

;
GET_HEX PROC NEAR                          ; 16進数を得る
    XOR     DX,DX                          ; 数をクリア

;
GET_HEX_LOOP:
    MOV     AL,[DI]                        ; 1桁ずつ切り出す
    CMP     AL,'a'                         ; 英子文字か判断する
    JB     ADJUST_NUM                     ; 英大文字か数字

;
    CMP     AL,'z'                         ; 英子文字か判断する
    JA     ADJUST_NUM                     ; 英子文字でない

;
    SUB     AL,20H                         ; 英大文字へ変換

;
ADJUST_NUM:
    CMP     AL,'0'                         ; 0以下か?
    JB     GET_HEX_EXIT                   ; 0以下であった

;
    CMP     AL,'9'                         ; 9以上か?
    JBE     ADJUST_NUM_1                 ; 0~9の範囲である

;
    CMP     AL,'A'                         ; A未満か?
    JB     GET_HEX_EXIT                   ; Aより小さい

;
    CMP     AL,'F'                         ; F以上か?
    JA     GET_HEX_EXIT                   ; F以上であった

;

```

```

ADJUST_NUM_1:
    SUB     AL,'0'           ; バイナリ数へ補正
    CMP     AL,9             ; A~Fか?
    JBE     ADD_NUM          ; 0~9である
;
;     SUB     AL,7           ; 補正
;
ADD_NUM:
    CBW                     ; 8ビットを16ビットへ変換
    SHL     DX,1             ; DXを16倍する
    SHL     DX,1
    SHL     DX,1
    SHL     DX,1
    ADD     DX,AX
    INC     DI
    JMP     GET_HEX_LOOP
;
GET_HEX_EXIT:
    MOV     AX,DX            ; 結果をレジスタAXへ
    RET
GET_HEX ENDP
;
SET_HEX PROC NEAR           ; 4桁の16進数をバッファへセット
;     上位バイトをセット
    XCHG    AL,AH
    CALL    SET_HEX2
    ADD     DI,2
    XCHG    AL,AH           ; 下位バイトをセット
    CALL    SET_HEX2
    RET
SET_HEX ENDP
;
SET_HEX2 PROC NEAR          ; 2桁の16進数をバッファへセット
;     上位4ビットを取り出す
    PUSH    AX
    SHR     AL,1
    SHR     AL,1
    SHR     AL,1
    SHR     AL,1
    ADD     AL,'0'           ; ASCII文字へ変換
    CMP     AL,'9'           ; 0~9か?
    JBE     NOT_A_F
;
;     ADD     AL,7           ; A~Fへ補正
;
NOT_A_F:
    MOV     CS:[DI],AL       ; バッファへ格納
    POP     AX
    AND     AL,0FH           ; 下位4ビットを取り出す
    ADD     AL,'0'           ; ASCII文字へ変換
    CMP     AL,'9'           ; 0~9か?
    JBE     NOT_A_F2
;
;     ADD     AL,7           ; A~Fへ補正
;
NOT_A_F2:
    MOV     CS:[DI+1],AL     ; バッファへ格納
    RET
SET_HEX2 ENDP
;
DREG ENDP
;
CODE ENDS
;
END DREG

```

■図 3.2 DREG.COM ダンプリスト

```

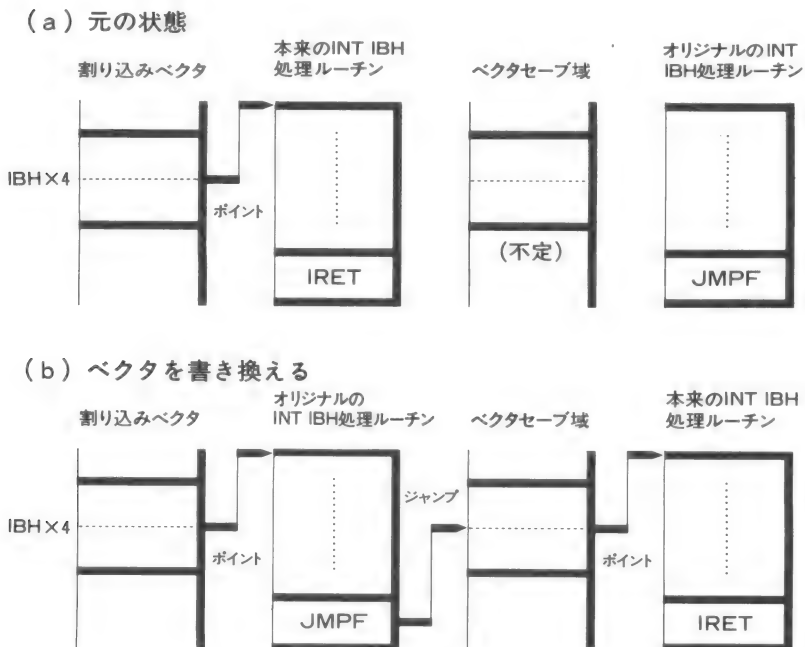
00000000 : 8D 1E 8C 01 33 F6 33 C9 83 F9 0A 74 65 8D 16 D0 : 72F
00000010 : 01 B4 09 CD 21 8D 16 8E 01 84 0A CD 21 80 3E BF : 637
00000020 : 01 00 74 4E 8D 3E C0 01 E8 99 03 89 40 03 8A 05 : 52E
00000030 : 3C 3A 74 0A 8D 16 03 02 84 09 CD 21 E8 CA 47 E8 : 62B
00000040 : 82 03 89 40 01 8A 05 3C 2C 75 E9 47 8A 05 3C 61 : 517
00000050 : 72 06 3C 7A 77 02 2C 20 3C 42 82 01 74 0C 3C 57 : 437
00000060 : 82 02 74 06 3C 44 82 04 75 CA 88 10 83 C6 05 41 : 5CA
00000070 : EB 96 84 25 80 05 8D 16 EE 02 CD 21 84 09 8D 16 : 6F0
00000080 : 21 02 CD 21 88 00 31 8A 00 01 CD 21 00 00 00 00 : 3A3
00000090 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 : 000
000000A0 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 : 000
000000B0 : 00 00 00 00 00 00 00 00 00 00 00 00 00 10 00 : 010
000000C0 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 : 000
000000D0 : 0D 0A 83 41 83 68 83 8C 83 58 82 F0 93 FC 97 CD : 815
000000E0 : 82 85 82 C4 89 BA 82 83 82 A2 28 53 45 47 4D 45 : 7B2
000000F0 : 4E 54 3A 4F 46 46 53 45 54 2C 42 2F 57 2F 44 29 : 433
00000100 : 3A 20 24 0D 0A 93 FC 97 CD 82 AA 90 83 82 85 82 : 780
00000110 : AD 82 0A 82 E8 82 DC 82 89 82 F1 81 49 81 49 07 : 8E0
00000120 : 24 0D 0A 43 4F 50 59 83 4C 81 58 82 CC 88 40 94 : 5CE
00000130 : 5C 82 F0 95 CF 8D 58 82 85 82 DC 82 85 82 BD 81 : 9A3
00000140 : 42 0D 0A 24 0D 0A 41 58 3D 30 30 30 20 42 58 : 2E4
00000150 : 3D 30 30 30 30 20 43 58 3D 30 30 30 20 44 58 : 371
00000160 : 3D 30 30 30 30 20 53 49 3D 30 30 30 20 44 49 : 363
00000170 : 3D 30 30 30 30 20 42 50 3D 30 30 30 20 53 50 : 36F
00000180 : 3D 30 30 30 30 20 49 50 3D 30 30 30 0D 0A 43 : 30D
00000190 : 53 3D 30 30 30 20 44 53 3D 30 30 30 30 20 45 : 369
000001A0 : 53 3D 30 30 30 20 53 53 3D 30 30 30 20 46 : 379
000001B0 : 4C 41 47 53 3D 30 30 30 3D 0D 0A 4D 45 4D 4F 52 : 388
000001C0 : 59 3A 20 30 30 30 30 3A 30 30 30 20 00 00 00 : 28D
000001D0 : 00 00 00 00 00 00 00 0A 00 00 00 00 00 00 00 : 017
000001E0 : 00 00 00 00 00 00 00 00 00 00 00 00 00 FB 2E : 129
000001F0 : A3 D8 02 2E 89 1E DA 02 2E 89 0E DC 02 2E 89 16 : 59E
00000200 : DE 02 2E 89 36 E0 02 2E 89 3E E2 02 2E 89 2E E4 : 651
00000210 : 02 2E 89 26 E6 02 2E 8C 1E E8 02 2E 8C 06 EA 02 : 535
00000220 : 2E 8C 16 EC 02 88 EC 8D 6E 06 2E A1 D8 02 8D 3E : 6AA
00000230 : 49 02 E8 C6 01 2E A1 DA 02 8D 3E 51 02 E8 88 01 : 667
00000240 : 2E A1 DC 02 8D 3E 59 02 E8 80 01 2E A1 DE 02 8D : 6A8
00000250 : 3E 61 02 E8 A5 01 2E A1 E0 02 8D 3E 69 02 E8 9A : 698
00000260 : 01 2E A1 E2 02 8D 3E 71 02 E8 8F 01 2E A1 E4 02 : 61F
00000270 : 8D 3E 79 02 E8 84 01 2E A1 E6 02 05 06 00 8D 3E : 540
00000280 : 81 02 E8 76 01 88 46 00 8D 3E 89 02 E8 6C 01 8D : 5EB
00000290 : 1E 44 02 89 48 00 E8 20 01 2E A1 E8 02 8D 3E 9A : 58F
000002A0 : 02 E8 57 01 2E A1 EA 02 8D 3E A2 02 E8 4C 01 2E : 5CF
000002B0 : A1 EC 02 8D 3E AA 02 E8 41 01 88 46 02 8D 3E 92 : 660
000002C0 : 02 E8 37 01 88 46 04 8D 3E B5 02 E8 2D 01 8D 1E : 53A
000002D0 : 8F 02 89 2C 00 E8 E1 00 8D 1E 8C 01 33 F6 33 C9 : 69C
000002E0 : 83 F9 0A 75 03 E9 99 00 2E 80 38 00 75 03 E9 90 : 657
000002F0 : 00 51 2E 88 40 03 8D 3E C3 02 E8 FE 00 2E 88 40 : 58C
00000300 : 01 8D 3E C8 02 E8 F3 00 53 2E 8A 00 2E C4 58 01 : 5C7
00000310 : FE C8 74 22 FE C8 74 3E 26 88 07 8D 3E D2 02 E8 : 813
00000320 : D9 00 2E C6 06 01 02 3A 26 88 47 02 8D 3E CD 02 : 574
00000330 : E8 C8 00 E8 3A 90 26 88 07 8D 3E CD 02 E8 C9 00 : 768
00000340 : 88 20 20 2E A3 CF 02 2E A3 D1 02 2E A3 D3 02 2E : 612
00000350 : A2 D5 02 E8 1A 90 26 88 07 8D 3E CD 02 E8 98 00 : 6E3
00000360 : 88 20 20 2E A3 D1 02 2E A3 D3 02 2E A2 D5 02 8D : 676
00000370 : 1E 88 02 89 1D 00 E8 40 00 83 C6 05 41 58 E9 5F : 608
00000380 : FF 2E A1 D8 02 2E 88 1E DA 02 2E 88 0E DC 02 2E : 62E
00000390 : 88 16 DE 02 2E 88 36 E0 02 2E 88 3E E2 02 2E 88 : 5E6
000003A0 : 2E E4 02 2E 8E 1E E8 02 2E 8E 06 EA 02 2E 8E 16 : 558
000003B0 : EC 02 2E 88 26 E6 02 FA CF 06 8C C8 8E C0 84 30 : 80A
000003C0 : CD 1A 07 C3 33 D2 8A 05 3C 61 72 06 3C 7A 77 02 : 589
000003D0 : 2C 20 3C 30 72 22 3C 39 76 08 3C 41 72 1A 3C 46 : 3CA
000003E0 : 77 16 2C 30 3C 09 76 02 2C 07 98 D1 E2 D1 E2 D1 : 6A8
000003F0 : E2 D1 E2 03 D0 47 E8 CE 88 C2 C3 86 C4 E8 09 00 : 983
00000400 : 83 C7 02 86 C4 E8 01 00 C3 50 D0 E8 D0 E8 D0 E8 : 98A
00000410 : 0D E8 04 30 3C 39 76 02 04 07 2E 88 05 58 24 0F : 42A
00000420 : 04 30 3C 39 76 02 04 07 2E 88 45 01 C3 : 2EB

```

## 3.2 ディスクアクセスのロギング

2.3 において触れたように、INT 1BH はチェックルーチンのありかを示しているのですが（もちろん正規に使用されている場合があります）、これでは、INT 1BH が存在する場所しかわかりません。いつ実行されるかまでは、プログラムを追跡してみない限りわからないのです。そこで、INT 1BH が実行される状況をプリンタへ打ち出すユーティリティ DLOG.COM を紹介します。

■図 3.3 ディスクアクセスのロギング



プログラムの原理は単純です。まず、割り込み番号 1BH に対応する割り込みベクタのアドレスを算出します。次にそのアドレスをどこかにセーブしておき、自らが用意する INT 1BH 処理ルーチンのアドレスを、代わりに設定しておきます。INT 1BH が実行されると制御は自分の中に移るわけですが、ここでスタックからレジスタ CS とレジスタ IP の値を取り出し、それらを 16 進数でプリンタへ出力します。また、レジスタの内容もプリンタへ出力します。そこで、セーブしておいた本来の INT 1BH 処理ルーチンのアドレスへジャンプするのです。もちろん、アドレスやレジスタの内容の表示において、レジスタの内容は保存しておきます。このようにを図 3.3 として示しました。

DLOG の詳しい使用法を説明します。まず、何もパラメータを与えずに DLOG を起動します。すると”INT 1BH”が実行された際に、出力したいメモリの位置を聞いてきますから、アドレスとサイズを DREG と同様の手順で入力してください。

ために、DLOG を用いて日本語ワープロ『松 86』を実行させてみました。図 3.4 として示しておきますので参考にしてください。

#### ■図 3.4 『松 86』のディスクアクセスを見る

```

A>DLOG [F] -----DLOG を常駐させる

アドレスを入力して下さい (SEGMENT:OFFSET,B/W/D): 0000:0014,D |-----COPY, STOP キー
アドレスを入力して下さい (SEGMENT:OFFSET,B/W/D): 0000:0018,D |のベクタをみてみる
アドレスを入力して下さい (SEGMENT:OFFSET,B/W/D):
ディスクロギング機能を付加しました。

A>MATU [F] -----松86を起動

AX=D690 BX=0400 CX=0307 DX=0103 SI=D690 DI=0010 BP=0010 SP=047A IP=4D8E
CS=0060 DS=0060 ES=44F6 SS=0646 FLAGS=FA02
MEMORY: 0000:0014 0060:51CE -----CS = 0060H だから IO, SYS から行われているものとみれる
MEMORY: 0000:0018 0AB3:2670

AX=D690 BX=0400 CX=0300 DX=0002 SI=D690 DI=0010 BP=0010 SP=0464 IP=4D8E
CS=0060 DS=0060 ES=44B5 SS=0646 FLAGS=FA02

```

MEMORY: 0000:0014 0060:51CE  
 MEMORY: 0000:0018 0AB3:2670

AX=D690 BX=0400 CX=0307 DX=0104 SI=D690 DI=0010 BP=0010 SP=047C IP=4D8E  
 CS=0060 DS=0060 ES=0646 FLAGS=FA02

MEMORY: 0000:0014 0060:51CE  
 MEMORY: 0000:0018 0AB3:2670

AX=D690 BX=0C00 CX=030F DX=0101 SI=D690 DI=0010 BP=0010 SP=047A IP=4D8E  
 CS=0060 DS=0060 ES=5D12 SS=0646 FLAGS=FA02

MEMORY: 0000:0014 0060:51CE  
 MEMORY: 0000:0018 0AB3:2670

AX=D690 BX=0400 CX=030F DX=0104 SI=D690 DI=0000 BP=0000 SP=0484 IP=4D8E  
 CS=0060 DS=0060 ES=AFB0 SS=0646 FLAGS=FA02

MEMORY: 0000:0014 5D32:086A  
 MEMORY: 0000:0018 5D32:0874 | COPY, STOP キーが変更を受けた

AX=D690 BX=0400 CX=030F DX=0105 SI=D690 DI=0010 BP=0010 SP=0480 IP=4D8E  
 CS=0060 DS=0060 ES=43F2 SS=0646 FLAGS=FA02

MEMORY: 0000:0014 5D32:086A  
 MEMORY: 0000:0018 5D32:0874

AX=0490 BX=006C CX=0000 DX=E090 SI=E016 DI=0100 BP=4600 SP=F9CE IP=ECBF  
 CS=5D32 DS=4D22 ES=0060 SS=4D22 FLAGS=F202

MEMORY: 0000:0014 5D32:086A  
 MEMORY: 0000:0018 5D32:0874 | CS:0060Hだから松86から直接INT 1BHが実行されたとみなせる

AX=D690 BX=0400 CX=0327 DX=0102 SI=D690 DI=0010 BP=0010 SP=0480 IP=4D8E  
 CS=0060 DS=0060 ES=43B1 SS=0646 FLAGS=FA02

MEMORY: 0000:0014 5D32:086A  
 MEMORY: 0000:0018 5D32:0874

AX=D690 BX=0400 CX=0327 DX=0103 SI=D690 DI=0010 BP=0010 SP=0480 IP=4D8E  
 CS=0060 DS=0060 ES=43B1 SS=0646 FLAGS=FA02

MEMORY: 0000:0014 5D32:086A  
 MEMORY: 0000:0018 5D32:0874

AX=D690 BX=0400 CX=0327 DX=0104 SI=D690 DI=0010 BP=0010 SP=0480 IP=4D8E  
 CS=0060 DS=0060 ES=43B1 SS=0646 FLAGS=FA02

MEMORY: 0000:0014 5D32:086A  
 MEMORY: 0000:0018 5D32:0874

AX=0490 BX=6F51 CX=0001 DX=0003 SI=0010 DI=F680 BP=4600 SP=F9FC IP=2A51  
 CS=5D32 DS=4D22 ES=0000 SS=4D22 FLAGS=F286

MEMORY: 0000:0014 5D32:086A  
 MEMORY: 0000:0018 5D32:0874

AX=5690 BX=0400 CX=0300 DX=0001 SI=6408 DI=F300 BP=0000 SP=F9F8 IP=2A51  
 CS=5D32 DS=4D22 ES=A800 SS=4D22 FLAGS=F246

MEMORY: 0000:0014 5D32:086A  
 MEMORY: 0000:0018 5D32:0874  
 BP=0000 SP=F9F6 IP=2A51

表示されるアドレスで、セグメントが 0060 であるものは IO.SYS から行われているもので、プロテクトとは無関係な場合が多いようです。セグメントがそれと異なる場合、プロテクトに使用されている可能性が大了。『松 86』の場合は、ハードディスクのリトラクト処理（ヘッドアームを無効シリンダへ移動させる処理）を行っているものも含まれています。

### ■図 3.5 DLOG.ASM ソースリスト

```

;*****
;
;      DLOG.ASM
;
;      INT 1BH発生による実行のレジスタ値・メモリ値の表示
;
;      COPYRIGHT(C) 1987 BY SHUWA SYSTEM TRADING CO.,LTD.
;
;      LAST MODIFIED ON FEBRUARY 2ND,1987
;*****
;
MEM_PACK      STRUC      ; アドレスセーブ領域の構造
SIZES          DB        ? ; 表示サイズ
OFFSETS        DW        ? ; 表示アドレスオフセット
SEGMENTS       DW        ? ; 表示アドレスセグメント
MEM_PACK      ENDS
;
DISK_VECT      EQU       1BH ; ディスクBIOS割り込みのベクタ番号
MAX_SAVE      EQU       10  ; 最大アドレスセーブ数
;
CODE          SEGMENT
ASSUME        CS:CODE,DS:CODE,ES:CODE,SS:CODE
;
;      ORG      100H
;
DLOG          PROC       NEAR
LEA           BX, MEM_INFO ; メモリ情報ブロックの先頭
XOR           SI, SI        ; メモリ情報ブロックのオフセット
XOR           CX, CX        ; メモリ情報ブロックのカウンタ
;
INPUT_ADDRESS:
CMP           CX, MAX_SAVE  ; 最大アドレス数を越えたか?
JE            INPUT_END    ; 越えたら入力を終了
;
LEA           DX, PROMPT    ; 入力を促すプロンプトのアドレス
MOV          AH, 9
INT          21H
;
LEA           DX, IN_BUFFER ; キーボードから入力
MOV          AH, 10
INT          21H
;
CMP           IN_BUFFER+1, 0 ; リターンキーのみの入力が見る
JE            INPUT_END    ; リターンキーのみの入力であった
;
LEA           DI, IN_BUFFER+2 ; 入力バッファの文字部のアドレス

```



```

CALL    GET_HEX          ; セグメントアドレスを取り出す
MOV     [BX+SI].SEGMENTS,AX ; セグメントアドレスをセット
MOV     AL,[DI]          ; 区切りがあるか見る
CMP     AL,';'           ; 区切りは正しいか?
JE      GOOD_DELIMIT     ; 区切りは正常であった
;
DISP_BAD_MESSAGE:
LEA     DX,BAD_INPUT     ; 入力不正であるというメッセージを表示
MOV     AH,9
INT     21H
JMP     INPUT_ADDRESS    ; 入力をもういちどやり直す
;
GOOD_DELIMIT:
INC     DI               ; オフセットアドレスを取り出す
CALL    GET_HEX
MOV     [BX+SI].OFFSETS,AX ; オフセットアドレスをセット
MOV     AL,[DI]          ; 区切りがあるか見る
CMP     AL,';'           ; 区切りは正しいか?
JNE     DISP_BAD_MESSAGE ; 区切りが正しくない
;
INC     DI               ; サイズ指定を取り出す
MOV     AL,[DI]          ; 1文字取り出す
CMP     AL,'a'           ; 英子文字か判断する
JB      ANALYSE_CHAR     ; 英大文字か数字
;
CMP     AL,'z'           ; 英子文字か判断する
JA      ANALYSE_CHAR     ; 英子文字でない
;
SUB     AL,20H           ; 英大文字へ変換
;
ANALYSE_CHAR:
CMP     AL,'B'           ; 1バイト表示の指示か?
MOV     DL,1             ; サイズを1バイトへ
JE      SET_SIZE         ; 1バイト表示の指示
;
CMP     AL,'W'           ; 2バイト表示の指示か?
MOV     DL,2             ; サイズを2バイトへ
JE      SET_SIZE         ; 2バイト表示の指示
;
CMP     AL,'D'           ; 4バイト表示の指示か?
MOV     DL,4             ; サイズを4バイトへ
JNE     DISP_BAD_MESSAGE ; 文字が不正
;
SET_SIZE:
MOV     [BX+SI].SIZES,DL ; サイズをセット
ADD     SI,SIZE MEM_PACK ; 次のアドレスへ
INC     CX               ; アドレス数を増す
JMP     INPUT_ADDRESS
;
INPUT_END:
MOV     AH,35H           ; 割り込みベクタ読み出し
MOV     AL,DISK_VECT     ; ディスクBIOSの割り込みベクタアドレス
INT     21H
MOV     WORD PTR VECTOR_SAVE,BX ; オフセットアドレスを待避
MOV     WORD PTR VECTOR_SAVE+2,ES ; セグメントを待避
;
MOV     AH,25H           ; 割り込みベクタ書き換え
MOV     AL,DISK_VECT
LEA     DX,ENTRY         ; 割り込み処理ルーチンのアドレス
INT     21H
MOV     AH,9             ; 変更した旨のメッセージを表示
LEA     DX,MESSAGE
INT     21H
;
MOV     AX,3100H         ; 常駐終了
MOV     DX,100H

```

```

INT 21H
;
VECTOR_SAVE DD ? ; 本来のINT 1BHベクタ内容退避領域
;
MEM_INFO MEM_PACK MAX_SAVE DUP(<0,0,0>)
;
IN_BUFFER DB 16 ; キー入力バッファ (最大文字数)
DB ? ; 実際に入力された文字数
DB 16 DUP(?) ; 文字部
;
PROMPT DB 13,10
DB 'アドレスを入力して下さい'
DB '(SEGMENT:OFFSET,B/W/D): '
DB '$'
;
BAD_INPUT DB 13,10
DB '入力が正しくありません!!'
DB 7,'$'
;
MESSAGE DB 13,10
DB 'ディスクロギング機能を付加しました。'
DB 13,10,'$'
;
SIZE_BUFFER1 = OFFSET OUT_BUFFER2-OFFSET OUT_BUFFER1
OUT_BUFFER1 DB 13,10,'AX=' ; レジスタ値出力用バッファ
OUT_AX DB '0000 BX='
OUT_BX DB '0000 CX='
OUT_CX DB '0000 DX='
OUT_DX DB '0000 SI='
OUT_SI DB '0000 DI='
OUT_DI DB '0000 BP='
OUT_BP DB '0000 SP='
OUT_SP DB '0000 IP='
OUT_IP DB '0000',13,10
;
SIZE_BUFFER2 = OFFSET OUT_BUFFER3-OFFSET OUT_BUFFER2
OUT_BUFFER2 DB 'CS=' ; セグメント値出力用バッファ
OUT_CS DB '0000 DS='
OUT_DS DB '0000 ES='
OUT_ES DB '0000 SS='
OUT_SS DB '0000 FLAGS='
OUT_FLAGS DB '0000',13,10
;
SIZE_BUFFER3 = OFFSET AXSAVE-OFFSET OUT_BUFFER3
OUT_BUFFER3 DB 'MEMORY: ' ; メモリ値出力用バッファ
OUT_SEGMENT DB '0000:'
OUT_OFFSET DB '0000 '
OUT_MEMORY DB 9 DUP(?)
DB 13,10
;
AXSAVE DW ? ; レジスタAXをセーブする領域
BXSAVE DW ? ; レジスタBXをセーブする領域
CXSAVE DW ? ; レジスタCXをセーブする領域
DXSAVE DW ? ; レジスタDXをセーブする領域
SISAVE DW ? ; レジスタSIをセーブする領域
DISAVE DW ? ; レジスタDIをセーブする領域
BPSAVE DW ? ; レジスタBPをセーブする領域
SPSAVE DW ? ; レジスタSPをセーブする領域
DSSAVE DW ? ; レジスタDSをセーブする領域
ESSAVE DW ? ; レジスタESをセーブする領域
SSSAVE DW ? ; レジスタSSをセーブする領域
;
ENTRY PROC FAR ; COPYキーが押下時の割込処理ルーチン
STI ; 割り込みを禁止する
MOV CS:AXSAVE,AX ; レジスタAXをセーブ
MOV CS:BXSAVE,BX ; レジスタBXをセーブ
MOV CS:CXSAVE,CX ; レジスタCXをセーブ

```

---

```

MOV     CS:DXSAVE,DX      ; レジスタDXをセーブ
MOV     CS:SISAVE,SI      ; レジスタSIをセーブ
MOV     CS:DISAVE,DI      ; レジスタDIをセーブ
MOV     CS:BPSAVE,BP      ; レジスタBPをセーブ
MOV     CS:SPSAVE,SP      ; レジスタSPをセーブ
MOV     CS:DSSAVE,DS      ; レジスタDSをセーブ
MOV     CS:ESSAVE,ES      ; レジスタESをセーブ
MOV     CS:SSSAVE,SS      ; レジスタSSをセーブ
;
;
MOV     BP,SP              ; スタックトップの値をBPへコピー
;
MOV     AX,CS:AXSAVE      ; レジスタの内容を表示(レジスタAX)
LEA     DI,OUT_AX
CALL    SET_HEX
;
MOV     AX,CS:BXSAVE      ; レジスタBXの内容をバッファへセット
LEA     DI,OUT_BX
CALL    SET_HEX
;
MOV     AX,CS:CXSAVE      ; レジスタCXの内容をバッファへセット
LEA     DI,OUT_CX
CALL    SET_HEX
;
MOV     AX,CS:DXSAVE      ; レジスタDXの内容をバッファへセット
LEA     DI,OUT_DX
CALL    SET_HEX
;
MOV     AX,CS:SISAVE      ; レジスタSIの内容をバッファへセット
LEA     DI,OUT_SI
CALL    SET_HEX
;
MOV     AX,CS:DISAVE      ; レジスタDIの内容をバッファへセット
LEA     DI,OUT_DI
CALL    SET_HEX
;
MOV     AX,CS:BPSAVE      ; レジスタBPの内容をバッファへセット
LEA     DI,OUT_BP
CALL    SET_HEX
;
MOV     AX,CS:SPSAVE      ; レジスタSPの内容をバッファへセット
ADD     AX,6               ; 割り込み発生に合せて補正
LEA     DI,OUT_SP
CALL    SET_HEX
;
MOV     AX,[BP]            ; レジスタIPの内容をバッファへセット
LEA     DI,OUT_IP
CALL    SET_HEX
;
LEA     BX,OUT_BUFFER1     ; 汎用レジスタの値をプリンタへ出力
MOV     CX,SIZE_BUFFER1
CALL    OUT_PRINTER
;
MOV     AX,CS:DSSAVE      ; レジスタDSの内容をバッファへセット
LEA     DI,OUT_DS
CALL    SET_HEX
;
MOV     AX,CS:ESSAVE      ; レジスタESの内容をバッファへセット
LEA     DI,OUT_ES
CALL    SET_HEX
;
MOV     AX,CS:SSSAVE      ; レジスタSSの内容をバッファへセット
LEA     DI,OUT_SS
CALL    SET_HEX
;
MOV     AX,[BP+2]         ; レジスタCSの内容をバッファへセット
LEA     DI,OUT_CS
CALL    SET_HEX

```

---

```

;
MOV     AX,[BP+4]      ; フラグレジスタの内容をバッファへセット
LEA     DI,OUT_FLAGS
CALL    SET_HEX
;
LEA     BX,OUT_BUFFER2 ; セグメントレジスタの値をプリンタへ出力
MOV     CX,SIZE_BUFFER2
CALL    OUT_PRINTER
;
LEA     BX,MEM_INFO    ; アドレス情報領域のアドレス
XOR     SI,SI          ; アドレス情報領域のオフセット
XOR     CX,CX          ; アドレス数をカウント
;
OUT_LOOP:
CMP     CX,MAX_SAVE    ; すべてのアドレスを出力したか?
JNE     OUT_LOOP1
JMP     ENTRY_EXIT     ; 出力したら終了
;
OUT_LOOP1:
CMP     CS:[BX+SI].SIZES,0 ; すべてのアドレスを出力したか?
JNE     OUT_LOOP2
JMP     ENTRY_EXIT     ; 出力したら終了
;
OUT_LOOP2:
PUSH    CX
MOV     AX,CS:[BX+SI].SEGMENTS ; セグメントをバッファへセット
LEA     DI,OUT_SEGMENT
CALL    SET_HEX
;
MOV     AX,CS:[BX+SI].OFFSETS ; オフセットをバッファへセット
LEA     DI,OUT_OFFSET
CALL    SET_HEX
;
PUSH    BX
MOV     AL,CS:[BX+SI].SIZES ; サイズで出力形式を変える
MOV     BX,CS:DWORD PTR [BX+SI].OFFSETS ; 出力対象アドレスを得る
DEC     AL                ; 1バイトの出力か?
JZ      OUT_BYTES        ; 1バイトの出力を行う
;
DEC     AL                ; 2バイトの出力か?
JZ      OUT_WORDS        ; 1ワードの出力を行う
;
MOV     AX,ES:[BX]        ; オフセット部を取り出す(前半)
LEA     DI,OUT_MEMORY+5
CALL    SET_HEX
MOV     CS:OUT_MEMORY+4,':' ; セパレータをバッファへセット
MOV     AX,ES:[BX+2]      ; セグメント部を取り出す(後半)
LEA     DI,OUT_MEMORY
CALL    SET_HEX
JMP     OUT_BUFFERS
;
OUT_BYTES:
MOV     AX,ES:[BX]        ; 内容を取り出す
LEA     DI,OUT_MEMORY
CALL    SET_HEX2
MOV     AX,2020H          ; 残りをスペースで埋める
MOV     CS:WORD PTR OUT_MEMORY+2,AX
MOV     CS:WORD PTR OUT_MEMORY+4,AX
MOV     CS:WORD PTR OUT_MEMORY+6,AX
MOV     CS:OUT_MEMORY+8,AL
JMP     OUT_BUFFERS
;
OUT_WORDS:
MOV     AX,ES:[BX]        ; 内容を取り出す
LEA     DI,OUT_MEMORY
CALL    SET_HEX
MOV     AX,2020H          ; 残りをスペースで埋める

```

```

MOV     CS,WORD PTR OUT_MEMORY+4,AX
MOV     CS,WORD PTR OUT_MEMORY+6,AX
MOV     CS,OUT_MEMORY+8,AL
;
OUT_BUFFERS:                                ; メモリ出力バッファの内容を出力
LEA     BX,OUT_BUFFER3
MOV     CX,SIZE_BUFFER3
CALL    OUT_PRINTER
ADD     SI,SIZE_MEM_PACK                    ; 次のアドレスへ
INC     CX
POP     BX
JMP     OUT_LOOP                            ; 最大アドレス数まで繰り返し
;
ENTRY_EXIT:
MOV     AX,CS:AXSAVE                        ; レジスタAXを復帰
MOV     BX,CS:BXSAVE                        ; レジスタBXを復帰
MOV     CX,CS:CXSAVE                        ; レジスタCXを復帰
MOV     DX,CS:DXSAVE                        ; レジスタDXを復帰
MOV     SI,CS:SISAVE                        ; レジスタSIを復帰
MOV     DI,CS:DISAVE                        ; レジスタDIを復帰
MOV     BP,CS:BPSAVE                        ; レジスタBPを復帰
MOV     DS,CS:DSSAVE                        ; レジスタDSを復帰
MOV     ES,CS:ESSAVE                        ; レジスタESを復帰
MOV     SS,CS:SSSAVE                        ; レジスタSSを復帰
MOV     SP,CS:SPSAVE                        ; レジスタSPを復帰
;
CLI                                           ; 割り込みを許可
JMP     CS:VECTOR_SAVE                     ; 本来の処理へ
ENTRY   ENDP
;
OUT_PRINTER PROC NEAR                      ; プリンタへの出力
PUSH    ES
MOV     AX,CS
MOV     ES,AX
MOV     AH,30H                             ; プリンタへのブロック出力
INT     1AH
POP     ES
RET
OUT_PRINTER ENDP
;
GET_HEX PROC NEAR                          ; 16進数を得る
XOR     DX,DX                              ; 数をクリア
;
GET_HEX_LOOP:
MOV     AL,[DI]                             ; 1桁ずつ切り出す
CMP     AL,'a'                             ; 英子文字か判断する
JB      ADJUST_NUM                          ; 英大文字か数字
;
CMP     AL,'z'                             ; 英子文字か判断する
JA      ADJUST_NUM                          ; 英子文字でない
;
SUB     AL,20H                             ; 英大文字へ変換
;
ADJUST_NUM:
CMP     AL,'0'                             ; 0以下か?
JB      GET_HEX_EXIT                       ; 0以下であった
;
CMP     AL,'9'                             ; 9以上か?
JBE     ADJUST_NUM_1                       ; 0~9の範囲である
;
CMP     AL,'A'                             ; A未満か?
JB      GET_HEX_EXIT                       ; Aより小さい
;
CMP     AL,'F'                             ; F以上か?
JA      GET_HEX_EXIT                       ; F以上であった
;

```

```

ADJUST_NUM_1:
    SUB     AL,'0'           ; バイナリ数へ補正
    CMP     AL,9             ; A~Fか?
    JBE     ADD_NUM          ; 0~9である
;
    SUB     AL,7             ; 補正
;
ADD_NUM:
    CBW                     ; 8ビットを16ビットへ変換
    SHL     DX,1             ; DXを16倍する
    SHL     DX,1
    SHL     DX,1
    SHL     DX,1
    ADD     DX,AX
    INC     DI
    JMP     GET_HEX_LOOP
;
GET_HEX_EXIT:
    MOV     AX,DX            ; 結果をレジスタAXへ
    RET
GET_HEX ENDP
;
SET_HEX PROC    NEAR        ; 4桁の16進数をバッファへセット
    XCHG    AL,AH            ; 上位バイトをセット
    CALL    SET_HEX2
    ADD     DI,2
    XCHG    AL,AH            ; 下位バイトをセット
    CALL    SET_HEX2
    RET
SET_HEX ENDP
;
SET_HEX2 PROC    NEAR        ; 2桁の16進数をバッファへセット
    PUSH    AX
    SHR     AL,1             ; 上位4ビットを取り出す
    SHR     AL,1
    SHR     AL,1
    SHR     AL,1
    ADD     AL,'0'           ; ASCII文字へ変換
    CMP     AL,'9'           ; 0~9か?
    JBE     NOT_A_F
;
    ADD     AL,7             ; A~Fへ補正
;
NOT_A_F:
    MOV     CS:[DI],AL       ; バッファへ格納
    POP     AX
    AND     AL,0FH           ; 下位4ビットを取り出す
    ADD     AL,'0'           ; ASCII文字へ変換
    CMP     AL,'9'           ; 0~9か?
    JBE     NOT_A_F2
;
    ADD     AL,7             ; A~Fへ補正
;
NOT_A_F2:
    MOV     CS:[DI+1],AL     ; バッファへ格納
    RET
SET_HEX2 ENDP
;
DLOG ENDP
;
CODE ENDS
;
END DLOG

```

■図 3.5 DLOG.COM ダンプリスト

```

00000000 : 8D 1E 9E 01 33 F6 33 C9 83 F9 0A 74 65 8D 16 E2 : 753
00000010 : 01 84 09 CD 21 8D 16 D0 01 84 0A CD 21 80 3E D1 : 658
00000020 : 01 00 74 4E 8D 3E D2 01 E8 B2 03 89 40 03 8A 05 : 659
00000030 : 3C 3A 74 0A 8D 16 15 02 B4 09 CD 21 EB CA 47 E8 : 63D
00000040 : 98 03 89 40 01 8A 05 3C 2C 75 E9 47 8A 05 3C 61 : 530
00000050 : 72 06 3C 7A 77 02 2C 20 3C 42 B2 01 74 0C 3C 57 : 437
00000060 : 82 02 74 06 3C 44 B2 04 75 CA 88 10 83 C6 05 41 : 5CA
00000070 : E8 96 84 35 80 1B CD 21 89 1E 9A 01 8C 06 9C 01 : 694
00000080 : 84 25 80 1B 8D 16 06 03 CD 21 B4 09 8D 16 33 02 : 4D3
00000090 : CD 21 B8 00 31 BA 00 01 CD 21 00 00 00 00 00 00 : 380
000000A0 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 : 000
000000B0 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 : 000
000000C0 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 : 000
000000D0 : 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 : 010
000000E0 : 00 00 0D 0A 83 41 83 68 83 8C 83 58 82 F0 93 FC : 6B1
000000F0 : 97 CD 82 B5 82 C4 89 BA 82 B3 82 A2 28 53 45 47 : 884
00000100 : 4D 45 4E 54 3A 4F 46 46 53 45 54 2C 42 2F 67 2F : 458
00000110 : 44 29 3A 20 24 0D 0A 93 FC 97 CD 82 AA 90 83 82 : 6E6
00000120 : B5 82 AD 82 A0 82 E8 82 DC 82 89 82 F1 81 49 81 : 9C7
00000130 : 49 07 24 0D 0A 83 66 83 42 83 58 83 4E 83 8D 83 : 578
00000140 : 4D 83 93 83 4F 88 40 94 5C 82 F0 95 74 89 C1 82 : 837
00000150 : B5 82 0C 82 B5 82 8D 81 42 0D 0A 24 0D 0A 41 58 : 637
00000160 : 3D 30 30 30 30 20 42 58 3D 30 30 30 30 20 43 58 : 36F
00000170 : 3D 30 30 30 30 20 44 58 3D 30 30 30 30 20 53 49 : 372
00000180 : 3D 30 30 30 30 20 44 49 3D 30 30 30 30 20 42 50 : 359
00000190 : 3D 30 30 30 30 20 53 50 3D 30 30 30 30 20 49 50 : 376
000001A0 : 3D 30 30 30 30 0D 0A 43 53 3D 30 30 30 30 20 44 : 30B
000001B0 : 53 3D 30 30 30 30 20 45 53 3D 30 30 30 30 20 53 : 378
000001C0 : 53 3D 30 30 30 30 20 46 4C 41 47 53 3D 30 30 30 : 3AA
000001D0 : 30 0D 0A 4D 45 4D 4F 52 59 3A 20 30 30 30 30 3A : 374
000001E0 : 30 30 30 30 20 00 00 00 00 00 00 00 00 00 0D 0A : 0F7
000001F0 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 : 000
00000200 : 00 00 00 00 00 00 F8 2E A3 F0 02 2E 89 1E F2 02 : 487
00000210 : 2E 89 0E F4 02 2E 89 16 F6 02 2E 89 36 F8 02 2E : 595
00000220 : 89 3E FA 02 2E 89 2E FC 02 2E 89 26 FE 02 2E 8C : 63D
00000230 : 1E 0D 03 2E 8C 06 02 03 2E 8C 16 04 03 8B EC 2E : 362
00000240 : A1 F0 02 8D 3E 61 02 E8 CA 01 2E A1 F2 02 8D 3E : 702
00000250 : 69 02 E8 8F 01 2E A1 F4 02 8D 3E 71 02 E8 B4 01 : 683
00000260 : 2E A1 F6 02 8D 3E 79 02 E8 A9 01 2E A1 F8 02 8D : 6F5
00000270 : 3E 81 02 E8 9E 01 2E A1 FA 02 8D 3E 89 02 E8 93 : 6E4
00000280 : 01 2E A1 FC 02 8D 3E 91 02 E8 88 01 2E A1 FE 02 : 66C
00000290 : 05 06 00 8D 3E 99 02 E8 7A 01 8B 46 00 8D 3E A1 : 511
000002A0 : 02 E8 70 01 8D 1E 5C 02 89 4B 00 E8 24 01 2E A1 : 544
000002B0 : 00 03 8D 3E B2 02 E8 6B 01 2E A1 02 03 8D 3E BA : 51F
000002C0 : 02 E8 50 01 2E A1 04 03 8D 3E C2 02 E8 45 01 8B : 559
000002D0 : 46 02 8D 3E AA 02 E8 3B 01 8B 46 04 8D 3E CD 02 : 552
000002E0 : E8 31 01 8D 1E A7 02 89 2C 00 E8 E6 00 8D 1E 9E : 669
000002F0 : 01 33 F6 33 C9 83 F9 0A 75 03 E9 99 00 2E 80 38 : 68C
00000300 : 00 75 03 E9 90 00 51 2E 8B 40 03 8D 3E DB 02 E8 : 5CE
00000310 : 02 01 2E 8B 40 01 8D 3E E0 02 E8 F7 00 53 2E 8A : 594
00000320 : 00 2E C4 58 01 FE C8 74 22 FE C8 74 3E 26 8B 07 : 6D7
00000330 : 8D 3E EA 02 E8 DD 00 2E C6 06 E9 02 3A 26 8B 47 : 693
00000340 : 02 8D 3E E5 02 E8 CC 00 E8 3A 90 26 8B 07 8D 3E : 6A0
00000350 : E5 02 E8 CD 00 8B 20 20 2E A3 E7 02 2E A3 E9 02 : 70A
00000360 : 2E A3 E8 02 2E A2 ED 02 E8 1A 90 26 8B 07 8D 3E : 695
00000370 : E5 02 E8 9F 00 8B 20 20 2E A3 E9 02 2E A3 E8 02 : 6E0
00000380 : 2E A2 ED 02 8D 1E D3 02 B9 1D 00 E8 44 00 83 C6 : 68A
00000390 : 05 41 5B E9 5F FF 2E A1 F0 02 2E 8B 1E F2 02 2E : 6A2
000003A0 : 8B 0E F4 02 2E 8B 16 F6 02 2E 8B 36 F8 02 2E 8B : 5F8
000003B0 : 3E FA 02 2E 8B 2E FC 02 2E 8E 1E 00 03 2E 8E 06 : 4BE
000003C0 : 02 03 2E 8E 16 04 03 2E 8B 26 FE 02 FA 2E FF 2E : 512
000003D0 : 9A 01 06 8C C8 8E C0 B4 30 CD 1A 07 C3 33 D2 8A : 767
000003E0 : 05 3C 61 72 06 3C 7A 77 02 2C 20 30 30 72 22 3C : 3D1
000003F0 : 39 76 08 3C 41 72 1A 3C 46 77 16 2C 30 3C 39 76 : 3E6
00000400 : 02 2C 07 98 D1 E2 D1 E2 D1 E2 D1 E2 03 D0 47 E8 : 99E
00000410 : CE 8B C2 C3 86 C4 E8 09 00 83 C7 02 86 C4 E8 01 : 898
00000420 : 00 C3 50 D0 E8 D0 E8 D0 E8 D0 E8 04 30 3C 39 76 : 912
00000430 : 02 04 07 2E 88 05 58 24 0F 04 30 3C 39 76 02 04 : 278
00000440 : 07 2E 88 45 01 C3 : 1C6

```

## 3.3 システムコールのロギング

”INT 1BH”のロギングと同様に、システムコール（”INT 21H”、資料編を参照）をロギングしても効果があります。特に、メッセージの表示やファイルの入出力を行う位置がわかると、解読の際の参考になることが多いようです。

具体的な方法としては、ディスクアクセスのロギングと同様の手段で実現することができます。





## 応用編 I



## 応用編 I

1. プロテクト表現のテクニック
2. プログラムを読みにくくする
3. 目立つ命令をかくす
4. MS-DOS版プロテクト技法

応用編 I と名付けて「“もう 1 つのプロテクト”を施す側に立った」テクニックの数々、中でも基本的なものについて紹介します。これらは、なかば当然ともいえるテクニックばかりですが、目新しさもあるはずです。次の応用編 II に進まれる前に、一読されることをお勧めします。

なお、応用編 I、応用編 II で紹介されるさまざまなテクニックは、あくまでも独立したテクニックであり、それを用いたからといってどうなるというものではありません。しかし、これらを 3 個、4 個と組み合わせると効果は倍増します。一つひとつのテクニックではおもしろみのないものも、2 個、3 個と組み合わせればとたんにおもしろさは倍増します。これらを組み合わせたらどうなるかということらを常に頭の中に入れて、読み進んでいただきたいと思います。

## 1

## プロテクト表現のテクニック

プロテクトチェックは、正常でないディスクにおいてプログラムの実行を停止させるために行うのですが、ただ行えばよいというのではなく、行っていることをわからないようにするのも1つのテクニックといえます。ここでは、そのようなプロテクトチェックということがらについて触れていきます。

## 1.1 チェック→エラーは早すぎる

## ○考え方

プロテクトがかかっている場合、多くのソフトウェアではエラーを発見したとたんに、“システムエラー”などと表示して実行を停止してしまいますが、これではプロテクトがかけられているという事実が露見するばかりか、プロテクトをチェックしているタイミングを教えていることにもなってしまいます。何もプロテクトをチェックしたからといって、すぐにメッセージを出力する必要はないのです。要は、プロテクトチェックに失敗したことを覚えておけばよいのです。

## ○実現方法

具体的には、プログラムのどこかでチェックを行って、そのチェックの情報をメモリ上のどこかに格納しておきます。情報の内容に

対する値としては、

0 0 H	.....	未チェック
0 1 H	.....	チェック成功
0 2 H	.....	チェック失敗

などとしておけばよいでしょう。この情報を、必要なときに取り出して判断した結果、エラーメッセージなりを出力してやれば、プロテクトチェックを行うタイミングをつかみにくくする効果はあるでしょう。

これに対抗する手段として、ディスクアクセスのロギング、システムコールのロギングを行う方法があります。これですと、ディスクアクセスを検出すると、プリンタへその位置とレジスタの内容が表示されますが、このとき、エラーメッセージの出力が行われないうでしたら、その内容がどこかに保存され、あとで参照されている可能性があるわけです。

また、ディスクアクセスのロギングを利用して、プログラム中でディスクアクセスを行っている箇所をリストアップし、その周辺を逆アセンブルします。ディスクアクセスの結果をメモリ上に格納しているようすが見られたら、間違いなくあとで参照されると思ってよいでしょう。

### ○サンプル

ドライブBのフォーマットをチェックし、異常があれば約30秒のカウントを行った後に、エラーメッセージを出力するプログラムCHKWAIT.COMを実行例とともに図1.1に示します。ドライブBには、MS-DOSフォーマット以外のフロッピーディスクを入れておきます（たとえばDISK BASICのものなどを入れる）。

CHKWAIT.COMにはパラメータはありませんので、コマンド

名のみを入力してから実行してください。

■図 1.1 CHKWAIT.ASM ソースリスト

```

;
;*****
;
;      CHKWAIT.ASM
;
;      ドライブ B のフォーマットをチェックし、
;      一定時間経過後にエラーメッセージを出力
;
;      COPYRIGHT(C) 1987 BY SHUWA SYSTEM TRADING CO.,LTD.
;
;      LAST MODIFIED ON FEBRUARY 3RD,1987
;*****
;
DRIVE      =          1          ; ドライブ B をチェック
;
IOSYS      SEGMENT AT 0060H      ; I O . S Y S のセグメントを定義
;
;      ORG          6CH
;
PDA_TABLE      LABEL   BYTE      ; ドライブに対応したPDAの配列
;
IOSYS      ENDS
;
CODE      SEGMENT
;      ASSUME      CS:CODE,DS:CODE,ES:IOSYS,SS:CODE
;
;      ORG          100H
;
CHKWAIT PROC
MOV         AX,IOSYS              ; レジスタESをセグメントIOSYSに設定
MOV         ES,AX
;
MOV         AL,PDA_TABLE+DRIVE    ; ドライブに対応したPDAを取り出す
MOV         DL,AL
AND         AL,0F0H               ; 装置番号を除く
MOV         CH,3                  ; セクタ長を3へ(1MBディスク用)
CMP         AL,90H                ; 1MBディスクか?
JE          CHECK
;
MOV         CH,2                  ; セクタ長を2へ(640KBディスク用)
CMP         AL,70H                ; 640KBディスクか?
JE          CHECK
;
LEA         DX,BAD_DRIVE          ; ドライブ不正のメッセージを出力
MOV         AH,9
INT         21H
JMP         EXIT
;
;      ASSUME      ES:CODE          ; レジスタESをコードに戻す
;
CHECK:
MOV         AX,CS
MOV         ES,AX
MOV         AL,DL                 ; PDA
MOV         AH,56H                ; 倍密度データ読み出し
MOV         BX,400H               ; 読み出しデータ量
XOR         CL,CL                 ; シリンダ0
MOV         DH,CL                 ; ヘッド0
MOV         DL,1                  ; セクタ1
LEA         BP,BUFFER             ; 読み出し用バッファ

```

```

        INT     1BH
        JNC     EXIT
;
        MOV     CX,300          ; ダミーループの回数
;
DUMMY_LOOP:
        PUSH    CX
        XOR     CX,CX
        LOOP    $
        POP     CX
        DUMMY_LOOP
;
        LEA     DX,BAD_DISK    ; ディスクのフォーマットが
        MOV     AH,9           ; 異常である旨のメッセージを出力
        INT     21H
;
EXIT:
        MOV     AX,4C00H       ; 非常駐終了
        INT     21H
;
BUFFER DB 400H DUP(?)        ; 読み出し用バッファ
;
BAD_DRIVE DB 'ドライブの指定が違います。'
        DB 13,10,'$'
;
BAD_DISK DB 'ディスクが異常です。'
        DB 13,10,7,'$'
;
CHKWAIT ENDP
;
CODE     ENDS
;
        END     CHKWAIT

```

## ■図 1.1 CHKWAIT.COM ダンプリスト

```

00000000 : 88 60 00 8E C0 26 A0 6D 00 8A D0 24 F0 B5 03 3C : 6FB
00000010 : 90 74 11 85 02 3C 70 74 08 8D 16 55 05 84 09 CD : 57E
00000020 : 21 E8 2D 90 8C C8 8E C0 8A C2 B4 56 B8 00 04 32 : 782
00000030 : C9 8A F1 B2 01 8D 2E 55 01 CD 18 73 13 89 2C 01 : 65C
00000040 : 51 33 C9 E2 FE 59 E2 F8 8D 16 72 05 84 09 CD 21 : 825
00000050 : B8 00 4C CD 21 00 00 00 00 00 00 00 00 00 00 : 1F2
;
0060H~044Fまですべて00H
;
00000450 : 00 00 00 00 00 83 68 83 89 83 43 83 75 82 CC 8E : 591
00000460 : 77 92 E8 82 AA 88 E1 82 A2 82 DC 82 B7 81 44 0D : 913
00000470 : 0A 24 83 66 83 42 83 58 83 4E 82 AA 88 D9 8F ED : 791
00000480 : 82 C5 82 B7 81 44 0D 0A 07 24 : 387

```

## ■図 1.1 CHKWAIT.COM 実行例

```

A>CHKWAIT [F] _____ MS-DOSのディスクで実行、何も起こらない
A>CHKWAIT [F] _____ BASICのディスクで実行
ディスクが異常です。 _____ しばらく間があり表示される
A>

```

## 1.2 エラーを出すだけでは能がない

### ○考え方

プロテクトチェックに失敗したら、エラーメッセージや警告を出力して、プログラム本来の動作を停止させてしまうというのが一般的でしたが、しかし、何もご丁寧にチェックの失敗をユーザに知らせてやる必要はないのです。エラーメッセージが出力されるのは、ユーザがそこで使用をやめるよう、または、複製を諦めるようにしむけるためですが、ある意地の悪いソフトハウス（現在は存在しません）は、次のような手段に出たことがあります。

それは、PC-8801用のワープロソフトで、プロテクトチェックに失敗しても、表向きには正常に動作しているように見せるというものです。ユーザは安心し、てっきりコピーできているものと思い、そのワープロを使用していました。ところがある日、自分の作成した文書がまったくでたらめなのに気付きます。ワープロソフトがプロテクトチェックに失敗したのを憶えていて、表向きは正常に、しかし裏ではアブノーマルに振る舞っていたのです。一種の多重人格でしょうが、意地の悪いことこの上もありません。何しろ、実際に業務で使用してしまっているのですから。当時、このようなプロテクトは話題になったものですが、これには賛否両論があり、評価も人それぞれだったようです。

このソフトハウスのやり方は、一步間違えば保障問題にも発展しかねないものですが、このように、問題にされることもなくやんわりと、そして手厳しくユーザを非難したソフトハウスもありました。

あるPC-9801用のワープロソフトでは、プロテクトチェックに失敗すると文書印刷と文書保存ができない旨のメッセージを出力し、

通常の動作に移ります。ユーザは、これを見てコピーに失敗したことに気付くのですが、中にはソフトハウスに苦情をいった人もいたそうです（「文書の保存と印刷ができないとすると不良品なのではないでしょうか…」）。ワープロソフトにとって、必須ともいえる機能を使用することができないのですから、これではコピーされてもしかたありません。

### ○実現方法

1.1 と同じです。要するに、プロテクトチェックを行った際に、その結果を憶えておき、実際に仕事をするときにはその結果を参照するのです。プロテクトチェックに成功したら通常の処理を、失敗したら偽の処理を行えばよいのです。

### ○対処方法

まず偽の処理を行っていることを発見することです。発見したら、ディスクアクセスをロギングし、1.1 と同様にチェック箇所をリストアップし、周辺を逆アセンブルします。

### ○サンプル

ドライブ B のフォーマットをチェックし、異常がなければ TYPE コマンドと同様の動きを、異常があれば、どのようなファイルが指定されていようが IO.SYS を画面に表示するプログラム CHKTYPE.COM を、実行例と共に図 1.2 として示します。図 1.1 におけるものと同様に、ドライブ B には、MS-DOS フォーマット以外のフロッピーディスクを入れておきます。

CHKTYPE.COM には、TYPE コマンドと同様にファイルを 1 個だけ指定します。



■図 1.2 CHKTYPE.ASM ソースリスト

```

;
;*****
;
;      CHKTYPE.ASM
;
;      ドライブ B のフォーマットをチェックし、
;      TYPE コマンドの働きを変化させる
;
;      COPYRIGHT(C) 1987 BY SHUWA SYSTEM TRADING CO.,LTD.
;
;      LAST MODIFIED ON FEBRUARY 3RD,1987
;*****
;
DRIVE      =      1      ; ドライブ B をチェック
IOSYS      SEGMENT AT 0060H      ; I O . S Y S のセグメントを定義
;
;      ORG      6CH
;
PDA_TABLE      LABEL      BYTE      ; ドライブに対応したPDAの配列
;
IOSYS      ENDS
;
CODE      SEGMENT
ASSUME      CS:CODE,DS:CODE,ES:IOSYS,SS:CODE
;
;      ORG      80H
;
PARAM      LABEL      BYTE      ; パラメータエリア
;
;      ORG      100H
;
CHKTYPE     PROC
MOV         AX,IOSYS      ; レジスタESをセグメントIOSYSに設定
MOV         ES,AX
;
;      MOV         AL,PDA_TABLE+DRIVE      ; ドライブに対応したPDAを取り出す
;      MOV         DL,AL
;      AND         AL,0F0H      ; 装置番号を除く
;      MOV         CH,3      ; セクタ長を3へ(1MBディスク用)
;      CMP         AL,90H      ; 1MBディスクか?
;      JE          CHECK
;
;      MOV         CH,2      ; セクタ長を2へ(640KBディスク用)
;      CMP         AL,70H      ; 640KBディスクか?
;      JE          CHECK
;
;      LEA         DX,BAD_DRIVE      ; ドライブが不正である旨のメッセージを出力
;      MOV         AH,9
;      INT         21H
;      JMP         EXIT
;
;      ASSUME      ES:CODE      ; レジスタESをコードに戻す
;
CHECK:
MOV         AX,CS
MOV         ES,AX
MOV         AL,DL      ; PDA
MOV         AH,56H      ; 倍密度データ読み出し
MOV         BX,400H      ; 読み出しデータ量
XOR         CL,CL      ; シリンダ0
MOV         DH,CL      ; ヘッド0
MOV         DL,1      ; セクタ1
LEA         BP,BUFFER      ; 読み出し用バッファ

```

```

INT     1BH
LEA     AX,DEFAULT_NAME
JC      TYPE_FILE      ; エラー時にはデフォルトのファイルを指定
;
;
LEA     BX,PARAM        ; パラメータ行をファイル名へ変換
MOV     AL,PARAM        ; パラメータをASCIZ文字列へ変換
XOR     AH,AH
MOV     SI,AX
MOV     [BX+SI+1],AH
;
SKIP_LOOP:
INC     BX
CMP     BYTE PTR [BX],' ' ; 空白をスキップ
JE      SKIP_LOOP
;
LEA     AX,[BX]
;
TYPE_FILE:
; ファイルを出力
MOV     DX,AX            ; ファイルをオープン
MOV     AX,3D00H;
INT     21H
MOV     BX,AX
JNC     TYPE_FILE_1
;
LEA     DX,BAD_FILE      ; ファイルがオープンできない
MOV     AH,9             ;
INT     21H              ; メッセージを出力
JMP     EXIT
;
TYPE_FILE_1:
MOV     AH,3FH           ; ファイルから読み出し
MOV     CX,1             ; 1文字
LEA     DX,BUFFER        ; 読み出し用バッファ
INT     21H
OR      AX,AX
JZ      CLOSE_FILE      ; ファイルの末端ならばファイルをクローズ
;
MOV     AH,2             ; 1文字コンソールへ出力
MOV     DL,BUFFER
INT     21H
JMP     TYPE_FILE_1      ; 次の文字へ
;
CLOSE_FILE:
; ファイルをクローズ
MOV     AH,3EH
INT     21H
;
EXIT:
MOV     AX,4C00H         ; 非常駐終了
INT     21H
;
BUFFER DB 400H DUP(?)   ; 読み出し用バッファ
;
DEFAULT_NAME DB 'A:\IO.SYS',0 ; ディスク異常時の出力ファイル名
;
BAD_DRIVE DB 'ドライブ B: をチェックできません。'
DB 13,10,7,'$'
;
BAD_FILE DB 'ファイルが見つかりません。'
DB 13,10,7,'$'
;
CHKTYPE ENDP
;
CODE ENDS
;
END CHKTYPE

```

## ■図 1.2 CHKTYPE.COM ダンプリスト

```

00000000 : B8 60 00 8E C0 26 A0 6D 00 8A D0 24 F0 B5 03 3C : 6FB
00000010 : 90 74 11 85 02 3C 70 74 0B 8D 16 99 05 B4 09 CD : 5C2
00000020 : 21 EB 67 90 8C C8 8E C0 8A C2 B4 56 B8 00 04 32 : 7EC
00000030 : C9 8A F1 B2 01 8D 2E 8F 01 CD 1B 8D 06 8F 05 72 : 6C3
00000040 : 16 8D 1E 80 00 A0 80 00 32 E4 88 F0 88 60 01 43 : 61E
00000050 : 80 3F 20 74 FA 8D 07 8B D0 B8 00 3D CD 21 88 D8 : 782
00000060 : 73 0B 8D 16 BF 05 B4 09 CD 21 EB 1E 90 B4 3F B9 : 6D5
00000070 : 01 00 8D 16 8F 01 CD 21 0B C0 74 0A B4 02 8A 16 : 4C1
00000080 : 8F 01 CD 21 EB E7 B4 3E CD 21 B8 00 4C CD 21 00 : 722

```

0090H～047FHまですべて00H

```

00000480 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 41 : 041
00000490 : 3A 5C 49 4F 2E 53 59 53 00 83 68 83 89 83 43 83 : 598
000004A0 : 75 20 42 3A 20 82 F0 83 60 83 46 83 62 83 4E 82 : 687
000004B0 : C5 82 AB 82 DC 82 B9 82 F1 81 44 0D 0A 07 24 83 : 788
000004C0 : 74 83 40 83 43 83 8B 82 AA 8C A9 82 C2 82 A9 82 : 85D
000004D0 : E8 82 DC 82 B9 82 F1 81 44 0D 0A 07 24 : 5FB

```

## ■図 1.2 CHKTYPE.COM 実行例

```

A>CHKTYPE CONFIG.SYS [F] -----ドライブBにMS-DOSのディスクを入れて実行
files=10
buffers=20
DEVICE=MMTK.DRV A: /P /N -----正常に表示される
DEVICE=Gram2.sys
shell=command.com a:% /e:16 /p

```

```

A>CHKTYPE CONFIG.SYS [F] -----BASICのディスクを入れて実行
_KNJDIC SYSBA[~^f [;R6^C -----何やらわけのわからないものが表示された

```

A>

## 1.3 プロテクトの方法は 1つではない

### ■製品ごとに種類を変える

#### ○考え方

従来はプロテクトは1個だけと決っていたようですが（コストや工程の都合からでしょうが）、現在では複数のプロテクトを効果的に組み合わせるのが一般的となっています。現在もっとも進んだ私たちは、ハード的に最強のプロテクトに複雑なチェックルーチンを絡ませたものといえます。

ここで紹介するのは、このような意味合いの“1個”ということではないのです。極端なことをいえば、製品は1個だけ生産されるわけではないのですから、それぞれに異なるプロテクトを施してもかまわないのです。

#### ○実現方法

プロテクトを明らかに変化させるには、生産ラインに手を加える以外ありませんが、プログラムが自らに手を加えるというものも考えられます。つまり、ソフトウェアを1回も起動しなければコピーできるというものです。1回でも実行してしまったら、自らにプロテクトがかかり、次回からはチェックされるというものです。このとき、実行のタイミングで異なるプロテクトと、チェックルーチンが有効になるように設定しておけば、生産ラインに手を加えることなく、複数のプロテクトを実現することができます。

## ○サンプル

プログラム起動時に時刻を参照し、0時から23時のどこに位置するかによって、4種類の異なるプロテクトをドライブB内のフロッピーディスクに施すプログラム PROTIME.COM を、図 1.3 として示します。プロテクトは、MS-DOS で使用されないトラックにかけられます (1MB ディスクでシリンダ 77 のヘッド 0、640KB ディスクでシリンダ 80 のヘッド 0)。

PROTIME.COM は、パラメータを与えずにコマンド名のみを指定します。また、かけられるプロテクトの種類については、前作を参考に各自解析してください。

■図 1.3 PROTIME.ASM ソースリスト

```

;
;*****
;
;      PROTIME.ASM
;
;      時間をチェックし、ドライブ B の使用されないトラックに
;      異なった種類のフォーマットを施す
;
;      COPYRIGHT(C) 1987 BY SHUWA SYSTEM TRADING CO.,LTD.
;
;      LAST MODIFIED ON FEBRUARY 3RD,1987
;*****
;
DRIVE  =      1      ; ドライブ B にプロテクトを施す
;
IOSYS  SEGMENT AT 0060H      ; I O . S Y S のセグメントを定義
;
;      ORG      6CH
;
PDA_TABLE LABEL BYTE      ; ドライブに対応したPDAの配列
;
IOSYS  ENDS
;
CODE   SEGMENT
ASSUME CS:CODE,DS:CODE,ES:IOSYS,SS:CODE
;
;      ORG      100H
;
PROTIME PROC
MOV     AX,IOSYS      ; レジスタESをセグメントIOSYSに設定
MOV     ES,AX
;
;      MOV     AL,PDA_TABLE+DRIVE      ; ドライブに対応したPDAを取り出す
;      MOV     PDA,AL
;      AND     AL,0F0H      ; 装置番号を除く
;      MOV     CL,77      ; プロテクトトラックを77へ(1MBディスク用)

```

```

CMP     AL,90H           ; 1MBディスクか?
JE      CHECK_TIME

;

MOV     CL,80             ; プロテクトトラックを80へ(640KBディスク用)
CMP     AL,70H           ; 640KBディスクか?
JE      CHECK_TIME

;

LEA     DX,BAD_DRIVE     ; ドライブが不正である旨のメッセージを出力
MOV     AH,9
INT     21H
JMP     EXIT

;

ASSUME  ES:CODE          ; レジスタESをコードに戻す

;
CHECK_TIME:
MOV     AX,CS             ; 時刻をチェックする
MOV     ES,AX
MOV     TRACK,CL          ; プロテクトトラックをセット
MOV     AH,2CH           ; 時刻を読み出す
PUSH    DX
INT     21H
POP     DX
AND     CH,11B           ; 0~3時へ補正
XOR     CL,CL            ; プロテクトルーチンへジャンプ
XCHG    CL,CH            ; アドレスを計算
MOV     SI,CX
SHL     SI,1
CALL    PROC_TABLE[SI]

;
EXIT:
MOV     AX,4C00H         ; 非常駐終了
INT     21H

;
PROTECT1 PROC NEAR       ; セクタ長を変化させるプロテクトを施す
LEA     DI,BUFFER        ; IDを作る
MOV     AL,TRACK         ; C
STOSB
XOR     AL,AL            ; H
STOSB
INC     AL               ; R
STOSB
MOV     AL,4             ; N
STOSB

;

MOV     AH,5DH           ; フォーマットコマンド
MOV     AL,PDA
MOV     BX,4             ; IDバッファのサイズ
MOV     CH,4             ; セクタ長=4(2048)
MOV     CL,TRACK         ; フォーマットトラック
MOV     DH,0             ; フォーマットヘッド=0
MOV     DL,OFFH          ; フォーマット用データ
LEA     BP,BUFFER        ; IDバッファのアドレス
INT     1BH
RET

PROTECT1 ENDP

;
PROTECT2 PROC NEAR       ; 単密度フォーマットを施す
LEA     DI,BUFFER        ; IDを作る
MOV     AL,TRACK         ; C
STOSB
XOR     AL,AL            ; H
STOSB
INC     AL               ; R
STOSB
MOV     AL,4             ; N

```

```

;
; STOSB
;
MOV     AH,1DH           ; フォーマットコマンド
MOV     AL,PDA
MOV     BX,4             ; IDバッファのサイズ
MOV     CH,4             ; セクタ長=4 (1024)
MOV     CL,TRACK         ; フォーマットトラック
MOV     DH,0             ; フォーマットヘッド=0
MOV     DL,OFFH          ; フォーマット用データ
LEA     BP,BUFFER        ; IDバッファのアドレス
INT     1BH
RET

PROTECT2 ENDP
;
PROTECT3 PROC NEAR      ; デリテッドデータを使用する
; IDを作る
LEA     DI,BUFFER
MOV     AL,TRACK         ; C
STOSB
XOR     AL,AL            ; H
STOSB
INC     AL               ; R
STOSB
MOV     AL,4             ; N
STOSB
;
MOV     AH,5DH           ; フォーマットコマンド
MOV     AL,PDA
MOV     BX,4             ; IDバッファのサイズ
MOV     CH,4             ; セクタ長=4 (2048)
MOV     CL,TRACK         ; フォーマットトラック
MOV     DH,0             ; フォーマットヘッド=0
MOV     DL,OFFH          ; フォーマット用データ
LEA     BP,BUFFER        ; IDバッファのアドレス
INT     1BH
;
MOV     AH,59H           ; ライトデリテッドデータコマンド
MOV     AL,PDA
MOV     BX,400H          ; 書き込むバイト数
MOV     CH,4             ; セクタ長=4
MOV     CL,TRACK         ; 書き込むトラック
MOV     DH,0             ; 書き込むヘッド
MOV     DL,1             ; 書き込むセクタ番号
LEA     BP,BUFFER        ; データバッファのアドレス
INT     1BH
;
RET
PROTECT3 ENDP
;
PROTECT4 PROC NEAR      ; IDと実際のセクタアドレスが異なる
; IDを作る
LEA     DI,BUFFER
MOV     AL,TRACK         ; C
SHR     AL,1             ; 実際のCの半分の値を設定
STOSB
MOV     AL,1             ; H (反転させる)
STOSB
XOR     AL,AL            ; R (セクタ番号0)
STOSB
MOV     AL,4             ; N (これは実際と同じ)
STOSB
;
MOV     AH,5DH           ; フォーマットコマンド
MOV     AL,PDA
MOV     BX,4             ; IDバッファのサイズ
MOV     CH,4             ; セクタ長=4 (2048)
MOV     CL,TRACK         ; フォーマットトラック
MOV     DH,0             ; フォーマットヘッド=0

```

```

MOV     DL,OFFH           ; フォーマット用データ
LEA     BP,BUFFER        ; IDバッファのアドレス
INT     1BH
;
RET
PROTECT4 ENDP
;
BUFFER DB 400H DUP(?)    ; フォーマット用バッファ
;
PDA     DB ?              ; フォーマットをかけるドライブのPDA
;
TRACK   DB ?              ; プロテクトをかけるトラック
;
BAD_DRIVE DB 'ドライブ B: にはプロテクトをかけられません。'
DB 13,10,'$'
;
PROC_TABLE DW PROTECT1    ; プロテクトルーチンの
DW PROTECT2              ; ジャンプテーブル
DW PROTECT3
DW PROTECT4
;
PROTIME ENDP
;
CODE    ENDS
;
END      PROTIME

```

### ■図 1.3 PROTIME.COM ダンプリスト

```

00000000 : B8 60 00 8E C0 26 A0 6D 00 A2 09 06 24 F0 B1 4D : 65C
00000010 : 3C 90 74 11 B1 50 3C 70 74 08 8D 16 08 06 B4 09 : 4EE
00000020 : CD 21 EB 1E 90 8C C8 8E C0 88 0E 0A 06 B4 2C 52 : 701
00000030 : CD 21 5A 80 E5 03 32 C9 86 CD 88 F1 D1 E6 FF 94 : 9C4
00000040 : 3A 06 B8 00 4C CD 21 8D 3E 09 02 A0 0A 06 AA 32 : 494
00000050 : C0 AA FE C0 AA B0 04 AA B4 5D A0 09 06 B8 04 00 : 7AF
00000060 : 85 04 8A 0E 0A 06 B6 00 B2 FF 8D 2E 09 02 CD 18 : 576
00000070 : C3 8D 3E 09 02 A0 0A 06 AA 32 C0 AA FE C0 AA B0 : 7A7
00000080 : 04 AA B4 1D A0 09 06 B8 04 00 B5 04 8A 0E 0A 06 : 44E
00000090 : B6 00 B2 FF 8D 2E 09 02 CD 18 C3 8D 3E 09 02 A0 : 64E
000000A0 : 0A 06 AA 32 C0 AA FE C0 AA B0 04 AA B4 5D A0 09 : 7D6
000000B0 : 06 B8 04 00 B5 04 8A 0E 0A 06 B6 00 B2 FF 8D 2E : 548
000000C0 : 09 02 CD 18 B4 59 A0 09 06 B8 00 04 B5 04 8A 0E : 4BF
000000D0 : 0A 06 B6 00 B2 01 8D 2E 09 02 CD 18 C3 8D 3E 09 : 4BE
000000E0 : 02 A0 0A 06 D0 E8 AA B0 01 AA 32 C0 AA B0 04 AA : 769
000000F0 : B4 5D A0 09 06 B8 04 00 B5 04 8A 0E 0A 06 B6 00 : 496
00000100 : B2 FF 8D 2E 09 02 CD 18 C3 00 00 00 00 00 00 : 422

```

0110H~04FFHまですべて00H

```

00000500 : 00 00 00 00 00 00 00 00 00 00 00 83 68 83 89 83 : 27A
00000510 : 43 83 75 20 42 3A 20 82 C9 82 CD 83 76 83 8D 83 : 71D
00000520 : 65 83 4E 83 67 82 F0 82 A9 82 AF 82 E7 82 EA 82 : 945
00000530 : DC 82 B9 82 F1 81 44 0D 0A 24 47 01 71 01 98 01 : 5E0
00000540 : DD 01 : 0DE

```



## ■図 1.3 PROTIME.COM 実行例

```

A>TIME 00:00 [F5] ----- 時刻を故意に設定

A>PROTIME [F5] ----- プロテクトをかける

A>SYMDEB [F5] ----- SYMDEB でどのようなプロテクトがかかっているかみる
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086]

-A [F5] ----- そのためのプログラムを入力
30B9:0100 MOV AH,4A
30B9:0102 MOV AL,91
30B9:0104 MOV DH,0
30B9:0106 MOV CL,4D
30B9:0108 INT 1B
30B9:010A
-G=100,10A [F5] ----- 実行
AX=0091 BX=0000 CX=044D DX=0001 SP=CE36 BP=0000 SI=0000 DI=0000
DS=30B9 ES=30B9 SS=30B9 CS=30B9 IP=010A NV UP DI PL NZ NA PO NC
30B9:010A 0473 ADD AL,73 ;'s'
-Q [F5] ----- とりあえず終了

A>TIME 01:00 [F5] ----- 時刻を変えてみる

A>PROTIME [F5] ----- 再びプロテクトをかける

A>SYMDEB [F5]
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086]

-A [F5]
30B9:0100 MOV AH,4A
30B9:0102 MOV AL,91
30B9:0104 MOV DH,0
30B9:0106 MOV CL,4D
30B9:0108 INT 1B
30B9:010A
-G=100,10A [F5] ----- 実行
AX=E091 BX=0000 CX=033F DX=0008 SP=CE36 BP=0000 SI=0000 DI=0000
DS=30B9 ES=30B9 SS=30B9 CS=30B9 IP=010A NV UP DI PL NZ NA PO CY
30B9:010A 0473 ADD AL,73 ;'s'
-Q [F5] ----- IDがないのだ、プロテクトが異なっている

A>

```

## ■チェックの方法を変える

## ○考え方

チェックの方法はいくらでも変化させることができます。もちろん生産ラインに手を加えないでですが、これには PC-9801 のカレンダー時計を用いるという方法があります。PC-9801 ではカレンダー時計を電源の入／切にかかわらず動作させておきますから、故意に

変化させない限りは、正しい日付と時刻が常に得られるわけです。そして、これを利用してチェックルーチンを変化させるのです。

具体的には、チェックルーチンのアドレスを並べたテーブルを用意して、読み出した日付と時刻に対応させたアドレスを取り出し、そこにジャンプします。日時で変化させるのは、何もチェックルーチンのほうではなく、チェックするプロテクトの場所を変化させてもよいし、またチェックをなくしてしまってもよいのです。

### ○対処方法

この方法ですと、プロテクトを解析しコピーした時点では使用できても、1時間後には使用できないということもありえます。また、そんなに急ではなくても、コピーをして他人の手に渡った時点で、使用不能になることも考えられるわけです。この方法でプロテクトをかけられたら、いつどのプロテクトとチェックルーチンが有効なのかわからないのですから、ディスクアクセスのロギングを行い、プロテクトのかかっているソフトをできるだけ小刻みに実行します。そのたびにディスクアクセスを行っているアドレスが異なるようでしたら、このプロテクトですから、結果を検討して周期を割り出し、すべてのパターンをチェックして、あとは通常のチェックルーチン潰しに移るしかありません。

また日時をシステムコールか BIOS を用いて読んでいれば、そこをロギングする方法もあります。基本的には日付を読んでもそれが一元化されれば、チェックするプロテクトの種類も一元化されるわけです。こちらのほうが確実でしょう。

## ○サンプル

カレンダー時計を読み取り、時刻別に異なるトラックをチェックするプログラム CHKTIME.COM を、図 1.4 として示します。ドライブ B には、MS-DOS フォーマット以外のフロッピーディスクを入れておきます。

CHKTIME.COM は、パラメータを与えずにコマンド名のみを指定します。

■図 1.4 CHKTIME.ASM ソースリスト

```

;
;*****
;
;      CHKTIME.ASM
;
;      時間をチェックし、ドライブ B の異なったトラックの
;      フォーマットをチェック
;
;      COPYRIGHT(C) 1987 BY SHUWA SYSTEM TRADING CO.,LTD.
;
;      LAST MODIFIED ON FEBRUARY 3RD,1987
;
;*****
;
DRIVE      =          1          ; ドライブ B をチェックする
;
IOSYS      SEGMENT AT 0060H      ; I O . S Y S のセグメントを定義
;
;      ORG          6CH
;
PDA_TABLE  LABEL   BYTE        ; ドライブに対応した PDA の配列
;
IOSYS      ENDS
;
CODE       SEGMENT
;      ASSUME      CS:CODE,DS:CODE,ES:IOSYS,SS:CODE
;
;      ORG          100H
;
CHKTIME    PROC
MOV         AX,IOSYS            ; レジスタ ES をセグメント IOSYS に設定
MOV         ES,AX
;
;      MOV         AL,PDA_TABLE+DRIVE      ; ドライブに対応した PDA を取り出す
;      MOV         PDA,AL
;      AND         AL,0F0H                ; 装置番号を除く
;      CMP         AL,90H                ; 1MB ディスクか？
;      JE          CHECK_TIME
;
;      CMP         AL,70H                ; 640KB ディスクか？
;      JE          CHECK_TIME
;
;      LEA         DX,BAD_DRIVE          ; ドライブが不正である旨のメッセージを出力
MOV         AH,9

```

```

        INT     21H
        JMP     EXIT

;
        ASSUME  ES:CODE          ; レジスタESをコードに戻す
;
CHECK_TIME:
        MOV     AX,CS            ; 時刻をチェックする
        MOV     ES,AX
;
CHECK_TIME_LOOP:
        MOV     AH,2CH          ; 時刻を読み出す
        INT     21H
        XCHG    CL,DH           ; 秒をトラック番号とし分をヘッド番号とする
        AND     DH,1            ; ヘッドを補正
        MOV     AH,5AH          ; 倍密度リードID
        MOV     AL,PDA
        INT     1BH
;
        MOV     AH,0BH          ; キーボードステータスチェック
        INT     21H
        OR      AL,AL           ; キーボードバッファに文字はあるか?
        JZ      CHECK_TIME_LOOP
;
EXIT:
        MOV     AX,4C00H        ; 非常駐終了
        INT     21H
;
PDA      DB     ?              ; IDをチェックするドライブのPDA
;
BAD_DRIVE DB     'ドライブ B: はチェックできません。'
          DB     13,10,'$'
;
CHKTIME ENDP
;
CODE     ENDS
;
        END      CHKTIME

```

#### ■図 1.4 CHKTIME.COM ダンプリスト

```

00000000 : 88 60 00 8E C0 26 A0 6D 00 A2 42 01 24 F0 3C 90 : 65E
00000010 : 74 0F 3C 70 74 08 8D 16 43 01 84 09 CD 21 EB 1D : 548
00000020 : 90 8C C8 8E C0 B4 2C CD 21 86 CE 80 E6 01 B4 5A : 8C9
00000030 : A0 42 01 CD 1B B4 08 CD 21 0A C0 74 E8 B8 00 4C : 6A2
00000040 : CD 21 00 83 68 83 89 83 43 83 75 20 42 3A 20 82 : 5E1
00000050 : CD 83 60 83 46 83 62 83 4E 82 C5 82 AB 82 DC 82 : 883
00000060 : 89 82 F1 81 44 0D 0A 24 : 32C

```



# 2

## プログラムを読みにくくする

プログラムはきれいに書くものだ、という考え方が一般化していますが、これは、プロテクトの世界では通用しません。きれいに書かれたプログラムは非常に読みやすいので、同時に追跡もされやすいのです。プログラムの実行を追跡するには基礎編でも紹介したように、プログラムを実際に行かせてトレースする方法と、流れを頭の中で想定して、プログラムリストを追う方法とがあります。

ここでは後者の方法に対抗して、プログラムリストを読めないようにする方法（とはいかないまでも読みにくくする方法）を紹介しましょう。

## 2.1 意味のない命令を多用する

### ■実行する上で意味のない命令

#### ○考え方

プログラムは、必要な命令を組み合わせて作成するのはもちろんですが、実行に差し支えない不要な命令を挿入するのも解析を混乱させるのに効果があります。ここで取り上げるのは、実行の結果、特定のメモリやレジスタ、フラグの内容を変化させず、時間のみを喰うというものです。

#### ○実現方法

たとえば、次のようなプログラムがあったとします。

```
NOP
NOP
NOP
MOV    AX,AX
OUT    FFH,AL
JMP    NEXT
NEXT: XCHG CX,CX
PUSH   BX
MOV    DX,DX
POP    BX
```

一見して、何か意味があるのだろうか？と思わせるプログラムです。しかしここで本当に意味がないのでは、それこそ意味がありません。意味がないようにして意味があり、意味があるようで実は意味がないというのが望ましいのです。ここでは本当に意味のないものとしましょう。

さて、順番に追っていけばわかると思いますが、先頭に3つ並んでいるのはNOP命令です。NOP命令があるとき、多くの人はそこを無視してしまいがちです。しかし、案外と意味のある場合もあります。次には、等しいレジスタ間での転送命令があり、すぐにI/OポートFFHへ出力を行っています。ここで正直に解析してきた人ならば、資料などを参照してFFHの機能を確認めるでしょう。しかしPC-9801では、FFHにはいまのところ機能が定義されていません（出力を行っても何も起こらない）。

さらに続く命令へのジャンプ命令が存在します。この場合、改めてジャンプする必要はないのですから、実行上は無駄といえます。続いて、等しいレジスタ間での交換命令、すぐにPOPするPUSH命令があります。ここでいえることは、この流れを通してレジスタ

値、フラグ値が変化しないということです。ここにあげた命令群は、よく錯乱のための手段として用いられますが、あくまでも時間稼ぎとしての強さしか持たないようです。これらの命令について説明しましょう。

## **NOP**

文字どおり何もしない命令です（No Operation の略）。リストを追う上では、何の意味も持たない場合が多いのですが、アドレス補正のためにアセンブラが自動的に挿入したり、周辺機器とのタイミング合わせのために挿入される場合も多いようです。ですから、いちがいに無意味だとはいえません。

NOP 命令は、内部的には”XCHG AX, AX”という命令で処理されています。それは 90H という命令コードからも明らかです。

## **MOV <REG>, <REG>**

あるレジスタから、同じレジスタへ値を転送する命令です。実行後のレジスタの値は変化しません。

## **XCHG <REG>, <REG>**

同じレジスタ間での交換命令です。もちろん実行後でもレジスタの値は変化しません。すでに説明したように、NOP 命令は内部的に XCHG AX,AX 命令と等価です。

## **JMP <次のアドレス>**

次のアドレスにジャンプするのですから、実行は継続します。

## **PUSH <REG>と POP <REG>を連続する**

同じレジスタ間で PUSH と POP を行えば、レジスタの内容は

変化しません。影響を及ぼすとすればスタックです。ですから完全に無意味であるとはいいきれません。

## IN, OUT

定義されていない（意味のない）I/O に対するアクセス命令は、実行したからといって何か起きるわけではありませんが、ハードの知識に弱い人には心理的な圧迫効果があります。

これらは連続するのではなく、意味のある命令群の中に何気なくちりばめるように挿入するのがよいでしょう。

## ○対処方法

いま読んでいる命令が、意味のあるものか意味のないものかは、プログラムの流れからカンで判断するしかありません。また、意味がないと思っても、単に意味が理解できていない場合もあります。したがって、なまじカンに頼るのも考えものです。

## ○サンプル

図 2.1 として示すプログラム QUEST1.ASM から、存在しなくても実行上特に差し支えないという命令を搜してください。

### ■図 2.1 QUEST1.ASM ソースリスト

```

:
: *****
:
: QUEST1.ASM
:
: 無意味な命令をさがすサンプル
:
: このプログラムは、キーボードから入力された行を大文字←→小文字
: 反転して表示するものです。
: このプログラムには、意識してコメントを付加していません。
: ですが、ソースリストであるという分だけ、わかりやすいはず。
:
: COPYRIGHT(C) 1987 BY SHUWA SYSTEM TRADING CO.,LTD.

```



```

;
;      LAST MODIFIED ON FEBRUARY 9TH,1987
;
;*****
;
DATA    SEGMENT
;
MESSAGE1    DB      13,10
              DB      'キーボードから何か文字列を入力して下さい。'
              DB      13,10
              DB      '終わりはリターンキーのみです。'
              DB      's'
;
BUFFER1     DB      64,?
              DB      64 DUP(?)
;
BUFFER2     DB      13,10
              DB      64 DUP(?)
;
DATA    ENDS
;
CODE    SEGMENT
        ASSUME  CS:CODE,DS:DATA,ES:DATA,SS:STACK
;
QUEST1  PROC
        MOV     AX,DATA
        MOV     DS,AX
        MOV     ES,AX
        CLD
QUEST1_0:
        NOP
        NOP
        NOP
        PUSH    AX
        MOV     AH,9
        MOV     DX,OFFSET MESSAGE1
        INT     33
        MOV     AH,10
        MOV     DX,OFFSET BUFFER1
        INT     33
        POP     AX
        MOV     AX,AX
        MOV     BX,OFFSET BUFFER1
        MOV     AL,[BX+1]
        OR      AL,AL
        JZ      EXIT
        MOV     CL,AL
        XOR     CH,CH
        XCHG    CL,AL
        LEA     SI,[BX+2]
        MOV     DI,OFFSET BUFFER2+2
        JMP     QUEST1_3
QUEST1_3:
        PUSH    DX
        LODSB
        XCHG    AX,AX
        CMP     AL,'A'
        JB     QUEST1_1
        CMP     AL,'Z'
        JA     QUEST1_1
        CMP     AL,'a'
        JB     QUEST1_1
        CMP     AL,'z'
        JB     QUEST1_1
        SUB     AL,20H
        JMP     QUEST1_1

```

---

```

QUEST1_2:
    ADD     AL,20H
QUEST1_1:
    STOSB
    POP     DX
    IN      AX,DX
    NOP
    LOOP    QUEST1_3
    MOV     AL,24H
    STOSB
    MOV     AH,9
    MOV     DX,OFFSET BUFFER2
    INT     33
    JMP     QUEST1_0

EXIT:
    MOV     AX,4C00H
    INT     21H

;
QUEST1 ENDP
;
CODE ENDS
;
STACK SEGMENT STACK
;
        DW      256 DUP (?)
;
STACK ENDS
;
END      QUEST1

```

---

## ■実行した結果意味のない命令

### ○考え方

前記のプログラムは、実行の結果が何も問われないものでしたが、これらは多少万能選手的な色合いが濃く、同時にくせが強いため、存在すればすぐに見つけられてしまうでしょう。そこで要求されるのが、その場その場に応じて意味があったり無意味であったりする命令です。実際、意味があるかないかはその流れから判断するしかなく、プログラムの組まれ方によってはかなりの錯乱術となります。

### ○実現方法

同様に例を示してみましょう。

```

XOR     SI,SI
LEA     BX, [BX+SI]

```

この場合、レジスタ SI の内容がクリアされるのみで、レジスタ BX の値は不変です。つまり、LEA 命令の存在する意味がないわけです。しかし、以降でレジスタ SI の内容が参照される場合もありますので、騙されないようにしましょう。同様に加数、減数が 0 である場合の ADD 命令、SUB 命令も、一見存在の意味が感じられず、引っ掛かりやすいといえます。同様に、シフトカウンタを 1 としたシフト・ローテート命令もこの範疇に入るでしょう。ここで考えられるだけの命令をあげてみましょう。

#### 加数、減数を 0 とした ADD, SUB 命令

”ADD AX,0”などのように、加数が 0 であればフラグのセットのみが行われるだけで、レジスタ AX の内容は変化しません。

#### 乗数、除数を 1 とした MUL, DIV 命令 (DIV 実行時には意味ありげ)

レジスタ BX を 1 として”MUL BX”などを行えば、結果は変化しません。しかし、レジスタ DX が 0 クリアされますので注意が必要です。また、同様に”DIV BX”とした場合は、結果は変化しませんが、被除数が 16 ビットで表しきれない場合には、1 で割ると結果も 16 ビットで表せませんので、除算エラーが発生し”INT 00H”が発生します。注意してください。

#### レジスタ CL を 1 とした、可変数シフト、ローテート命令

レジスタ CL を 1 として、”SHL AX, CL”などとしても、結果は”SHL AX,1”と変わりません。これは前に示した LEA 命令などと同様に、あとでレジスタ CL が参照されているかどうかを注意する必要があります。

### POP 値を 0 とした RET 命令

RET 命令には、リターンと同時にスタックを何レベル捨てるか、そのレベル数を指定できますが、これを 0 とした場合、通常の RET 命令と動作は変わりません。

### 条件を固定した後の条件分岐命令

CF をセットしてすぐに "JC XXXX" などとするのは、"JMP XXXX" と変わりませんが、レジスタなどと同様、あとで参照されることもありますので注意しなければなりません。

### 意味のないプリフィクス命令

メモリアクセスのない命令に、セグメントオーバーライドのプリフィクスを付加したり、LOCK プリフィクスを付加したりするのは意味がありません。たとえば、

```
CS:
MOV    AX,BX
```

のようになります。

### 同一ブロックにおけるブロック転送

結果的にまったく同じアドレスになるようにアドレスを計算し、ブロック転送を行えば、ブロックの内容は変化しません。このときアドレスは複雑な演算を経て求めるのもよいのですが、セグメントとオフセットをうまく調整し、結果的に同じ絶対アドレスを指すようにするのもよいでしょう。

これらは、連続させると意味がなくなりますので、何気なく散りばめていくのがコツです。

## ○サンプル

図 2.2 として示すプログラム QUEST2.ASM において、結局何が行われているのかを当ててください。答は実行させてみればわかります。

■図 2.2 QUEST2.ASM ソースリスト

```

:
:*****
:
:      QUEST2.ASM
:
:      プログラムが何をやるものか当てるサンプル
:
:      実行して見れば、何をしているかわかります。
:
:      このプログラムには、意識してコメントを付加していません。
:      ですが、ソースリストであるという分だけ、わかりやすいはず。
:
:      COPYRIGHT(C) 1987 BY SHUWA SYSTEM TRADING CO.,LTD.
:
:      LAST MODIFIED ON FEBRUARY 9TH,1987
:
:*****
:
CODE    SEGMENT
        ASSUME    CS:CODE,DS:CODE,ES:CODE,SS:CODE
:
:      ORG        100H
:
QUEST2  PROC
        XOR       AH,AH
        MOV       AL,3DH
        MOV       CX,AX
        SHR       CX,CL
        INC       CL
        XCHG      AL,AH
        LEA       BP,EXIT-14
        PUSH      BP
        POP       DX
        INT       21H
        MOV       ES,AX
        PUSHF
        POP       AX
        AND       AX,CX
        JNZ       EXIT
QUEST2_1:
        NOT       AX
        XOR       AH,AH
        NOT       AX
        CBW
        MOV       AH,40H
        MOV       BX,ES
        MOV       DX,80H
        SUB       AH,CL
        INT       21H
        JNAE      EXIT
        INC       AX
        MOV       DI,AX

```

---

```

        DEC     AX
        MOV     CL,AL
        JZ      EXIT
        ROR     AL,CL
        MOV     SI,AX
        MUL     DI
        SHR     DI,CL
        MOV     DL,[SI]
        REP     MOVSB
        ROL     AL,CL
        ROL     AL,CL
        ADD     AL,2
        XCHG    AL,AH
        INT     21H
        SHR     AX,1
        XCHG    CL,AH
        JMP     QUEST2-1
        DB      41H,3AH,5CH,43H,4FH,4EH,46H,49H,47H,2EH,53H,59H,53H,0
EXIT:    MOV     AH,'L'
        INT     21H
;
QUEST2  ENDP
;
CODE    ENDS
;
        END     QUEST2

```

---

## 2.2 定石をあえて破る

### ■定石は理解しやすい

#### ○考え方

定石は、プログラミングの先人たちが積み重ねてきた経験によって、最良とされてきたプログラミング上の技法です。よって、定石についての知識があるか、経験によって定石を知らず知らずのうちに身につけてきたプログラマであれば、自ら進んで定石を用いプログラムされている箇所を見れば、容易に理解することができます。

ここではこの定石をあえて破り、解読の際の障害にしようというものです。

### ○実現方法

定石を破るための一般的な方法などはありませんが、定石を知らずにプログラミングを行っていた時代を思い出せばよいでしょう。そのときの洗練されていないプログラム、汚いプログラム、へたくそなプログラムが方法のすべてです。といっても、このまま突き放すわけにもいきません。いくつか例を示しましょう。

#### <例1：メモリ値の交換>

まずは簡単なものからです。メモリに格納されている内容を2カ所で交換するには、レジスタ AX を介して3命令で実現できます。

```
MOV    AX, MEM1
XCHG   AX, MEM2
MOV     MEM2, AX
```

これでおしまいです。しかし、MEM1 と MEM2 の内容を2つのレジスタへ読み出し、レジスタにおける交換命令を実行して再びメモリへ戻せば、命令は複雑になります。

```
MOV     AX, MEM1
MOV     DX, MEM2
XCHG    AX, DX
MOV     MEM2, DX
MOV     MEM1, AX
```

さらに、レジスタを介さない方法として、

```
PUSH    MEM1
PUSH    MEM2
POP     MEM1
```

POP MEM2

ともできます。特に 3 番目の例の場合、PUSH 命令と POP 命令の間に何かしらの処理がはさまっていれば、交換が目的であるとは、すぐには気付かないでしょう。

### <例 2：テーブルによるジャンプ>

次は、レジスタ AL に入っている内容に対応したアドレスへのジャンプを行うというものです。次のようにすれば、きれいで簡単です。

```
CBW
SHL  AX,2
MOV  SI,AX
JMP  TABLE [SI]
```

TABLE は、アドレスが並べられている表です。たとえば次のように宣言されているとします。

```
TABLE DW JUMP1 ; AL = 0 の場合
      DW JUMP2 ; AL = 1 の場合
```

レジスタ AL に与えられた値が 1 であれば、JUMP2 へのジャンプが行われます。さて、ここで正直にレジスタ AL の内容を判断し、それに応じてジャンプを行う例を示しましょう。

```
DEC  AL
JS   JUMP1
DEC  AL
JS   JUMP2
.....
```



レジスタ AL の値を 1 個ずつ減らし、負になったら対応するジャンプ命令を実行します。人によっては、こちらのほうが直接的でよくわかるという人もいでしょう。ジャンプではなくメッセージのアドレスを求めるとしても同様です。

## ■遠回し式コーディング

### ○考え方

ある機能を実現するための再短距離はあえて捨てて、複雑な過程の上に機能を成立させるというものです。CPU の命令に関する知識を駆使した、かなりトリッキーなプログラミングが要求されます。

### ○実現方法

定石を破るのと同様、きれいに書こうとしなければよいのです。例をいくつか示しましょう。

#### <例 1：0 テストをする>

レジスタ、またはメモリの内容が 0 であるかどうかを確かめる方法には、単純な目的ながら実にいろいろあります。ここでは、レジスタやメモリの内容を破壊せずにテストを行う方法を示しましょう。

AND	AX,AX	; あまりにもおなじみ
OR	AX,AX	; 上に同じ
TEST	AX,AX	; 上に同じ
CMP	AX,0	; 教本どおり
ADD	AX,0	; 多少トリッキー
CMP	MEM,0	; 教本どおり
TEST	MEM,NOT 0	; 全ビット 0 で 0 (あたりまえ)

レジスタやメモリ値の破壊が許されるなら、次のような手段も使用することができます。

```
SUB    AX,0 ; 最初から 0 である場合に限り 0
DEC    AX   ; 結果は SF, CF に反映
```

このようにたんに 0 テストを行うだけでも、かなりの方法を使い分けられることがわかるでしょう。

### <例 2：ジャンプする>

ジャンプといえば、JMP 命令で容易に実現できますが、スタックの特性と RET 命令の動作を活かして、遠回しにジャンプを行います。

```
MOV     AX,1000H
PUSH    AX
```

.....

```
RET
```

ある程度場数をこなしている読者であれば、すぐにわかるでしょう。そうです、オフセットアドレス 1000H にジャンプしているのです。RET 命令実行の時点で、スタックトップには値 1000H が積まれていますから、このような動作をするのです。単に”JMP 1000H”としないことに意味があるわけです。CALL 命令も、同様の手順で実現することができます。

```
LEA     AX,RETURN
PUSH    AX
MOV     AX,1000H
```

```
PUSH  AX
RET
RETURN:
```

この場合、最初に戻ってアドレスをスタックに積んでいる点を除けば、オフセット 1000H へのジャンプと同様です。

### ＜例 3：0 クリアを行う＞

メモリの特定の領域を 0 で埋めつくすには、リピートプリフィクスとストリングプリミティブを用いる方法がよく知られています。たとえば、次のようにです。

```
LEA    DI,ADDRESS
MOV    AX,MEMSIZE
XOR    AL,AL
REP    STOSB
```

非常に簡単に、ADDRESS で始まる領域を MEMSIZE バイトだけ 0 で埋めることができます（レジスタ ES をセグメントベースとすることに注意）。しかし、ここで次のようにすると、プログラムは汚くなります。

```
MOV    DI,OFFSET ADDRESS
MOV    AX,MEMSIZE-2
XOR    AL,AL
MOV    ES:[DI],AL
MOV    SI,DI
INC    DI
REP    MOVSB
```

一見するとブロック転送ですが、最初にストアされた 0 を、次のアドレスへ転送しているだけです（レジスタ DS = ES とする）。

#### <例 4：加算を行う・和を求める>

レジスタ AX とレジスタ BX の和を計算するのはいとも簡単ですが、これをまわりくどくすれば、次のようにすることもできます。

```
- LOOP: INC    AX
          DEC    BX
          JNZ    - LOOP
```

また、再帰的にプログラムを実行し和を求めれば、かなり複雑に見せかけることもできます。たとえば、アドレス DATA から格納される 10 個の数（すべて 1 ワードの大きさ）を加算し、その和を求めるとします。

```

                                XOR    AX,AX        ; 和を格納
                                LEA     BX,DATA      ; データのアドレス
                                MOV     CX,10        ; データの個数
ADDITION: OR      CX,CX        ; 全データ加えたか？
                                JZ      EXIT
                                ADD     AX, [BX]
                                ADD     BX,3
                                DEC     CX
                                CALL    ADDITION    ; 続くデータを加える
EXIT:    ADD     SP,20         ; CALL によるスタック消費を破棄
```

このようにたわいのない仕事でも、複雑な処理を経て実行すれば、かなり読みにくくなることは必至です。

以上、遠回し式コーディングの例は尽きることがありませんが（要するに手のこんだ、汚いともいえるプログラムを書けばよい）、このへんにしておきましょう。できればもっと紹介したいのですが、残りはみなさん自身で研究してください。

## 2.3 意味のない分岐

### ■分岐は何回まで耐えられるか

#### ○考え方

必要のないところでも JMP 命令を用いたり、CALL 命令を用いたりしましょう。一般に、リストを追うとき分岐する箇所があれば、そこまでの解析はひとまず置いて、分岐先の解析を始めるでしょうから、分岐の度合いが激しくかつ頻度が激しいほど効果があります。あなたはどこまで精神的に耐えられますか？

#### ○実現方法

特に実現方法などというものはないので、分岐はなるべく分岐した直後、すなわちサブルーチンの先頭などで行うとよいでしょう。ごくふつうのレベルでは、6 回サブルーチンコールが続けば嫌気がさすそうです。

## ■RET命令が現れない

何回もサブルーチンコールが続いて、解説をそこに移して、いつまでも RET 命令が現れない、あとになって、ずいぶん長いサブルーチンだ、などと気付いても手後れです。何も RET 命令でサブルーチンから戻る必要はないのです。

## 2.4 未定義命令を使う

### ○考え方

未定義命令とは、CPU のマニュアルや解説書には記載されていないが、実行させてみると動作するという命令のことです。一般に、未定義命令には予期される動作というものがあります。たとえば、Z80CPU において一部のセカンドソースでは、

```
DD 7C      LD A,IXH
```

なる命令が動作します。一部の雑誌には紹介されましたが、もちろんマニュアルには記載されていません。これは“レジスタ IX の上位バイトを、レジスタ A にロードせよ”という命令です。これが存在することを予測させる理由は、マニュアルにきちんと記載されている

```
21 00 00    LD HL,0
DD 21 00 00  LD IX,0
```

という命令です。実は、後者の命令は前者の命令にコード“DD”を付けただけのものなのです。よって、レジスタ IX を使うにはレジ

スタ HL を用いた命令に“DD”を付けただけと推測できますから、

```
7C          LD  A,H
```

という命令に“DD”を付ければ、前述した動作も期待できます。

これは Z80 での例ですが、8086 でもこのような命令が存在します。すなわち、命令コードの仕組みから存在すると考えられる命令です。逆アセンブラでは、それらは逆アセンブルできませんから、解析の妨害にはなります。未定義命令は、主にセグメントレジスタやスタックに関する命令で、次のようなものがあります。

### POP CS

スタックからレジスタ CS へ値をポップする命令です (POP DS, POP ES, POP SS という一連の命令の存在から推測できます)。ということは、レジスタ CS の内容が変化しますから、プログラムの位置も変化するということで、不用意にこの命令を実行したなら間違いなく暴走します。しかし、変化した CS に対しじつまを合わせるように、別のプログラムを置いておけば、プログラムは暴走せず、実行を継続することができるのです。なお、V30 や 80286 では、別の機能を持つ命令が割り当てられています。

### MOV CS, r/m16

レジスタ CS に値を移す命令です。POP CS と同様に、使用法を誤れば間違いなく暴走します。r/m16 は、レジスタかメモリを表します。

### AAD <N>, AAM <N>

AAD/AAM という命令は、それぞれ乗算命令における補正効果を持つものです。これらの命令の持つ動作は、

AAD :  $AL \leftarrow AH * 10 + AL, AH \leftarrow 0$

AAM :  $AH \leftarrow AL / 10, AL \leftarrow AL \% 10$

となっており、それぞれに 10 という定数が絡んでいます。実は、この 2 つの命令の命令コードは、それぞれ、

AAD : D5 0A

AAM : D4 0A

となっています。もうお気づきでしょう。命令コードの 2 バイト目、すなわち 0AH とは、10 を意味しているのです。実験で確認しましたが、この値を変更すれば、命令のオペレーションを変えることができます。

実際に、この命令を扱うことのできるアセンブラ、逆アセンブラが存在します。それは、DISK BASIC の MON コマンド内のもので、2 バイト目が 0AH 以外であっても、命令モニタックを表示します。

他に、コードは違うが同じ動作をするという命令も存在します。たとえば、XLAT 命令のコードは D7H ですが、D6H でも同じ効果を得ることができます。D6H のほうはアセンブラも逆アセンブラも認識しませんから、このようなコードを用いるのも効果的です。

また、実行しても何も起こらない命令を用いるのも、一つの方法です（80286 においては不正命令実行の割り込みが発生する）。

## ○実現方法

もともとマニュアルに記載されておらず、かつ期待される動作をする命令を用いればよいのですから、そう難しいことはありません。ただし、そのような命令を探すのがなかなか面倒で、すべての



CPU で同じような命令が存在するとも限りませんから、むやみに使用するのはいかたまりではないかもしれません。

### ○対処方法

逆アセンブルリストの中に、逆アセンブルできない命令が現れたら、とりあえず命令コード表と見比べて、何か意味がありそうかどうかを調べてみましょう。ちなみに、DISK BASIC の内蔵モニタの逆アセンブラ（L コマンド）では、セグメント外 JMP、セグメント外 CALL、セグメント外 RET の各命令は逆アセンブルできませんので、リスト中には“??”と表示します。

## 2.5 ソフトウェア割り込みを使う

### ■BIOSを多用する

#### ○考え方

INT 命令は、主にシステムの用意するさまざまな機能を呼び出す目的で使用されます。たとえば、ディスク BIOS の“INT 1BH”、MS-DOS のシステムコールの“INT 21H”など、多数あります。これらは、その割り込みの持つ機能を知らなければ、何を行っているのかわかりません。したがって、これを多用することでかなりの時間稼ぎにはなります。

#### ○実現方法

要するに INT 命令を用いればよいのです。たとえばキーボードや CRT、グラフィックス、ディスクなどの処理に BIOS を用いれば

よいのです。しかし、BIOS の多用は、あまりマシンに対する知識のない人にのみ有効であることを忘れてはなりません。

### ○対処方法

マシンについての知識をつけるしかありません。少なくとも、割り込みのタイプと機能の対応ぐらいは、すぐにわかるようにしたほうがよいでしょう（キーボード、CRT 関係ならば 18H などという具合に）。

## ■自ら定義する

### ○考え方

割り当てられていない割り込みベクタに、独自の機能を定義して、わざわざそれを呼び出します。このようなものはふつうどんな解説書にも記載されていないはずですから、中身を解析するの手段に出るはずですが、ただし、CALL 命令で呼び出すよりは余程効果的でしょう。

### ○実現方法

ふつうにサブルーチンを設計するのと同様、割り込み処理ルーチンを設計し、割り込みベクタの空いているタイプにアドレスを設定してやります。もちろん、割り込み処理ルーチンは IRET 命令によって終るようにします。

### ○サンプル

加減乗除を行うサブルーチンを、"INT 0D0H"に定義するプログラム INTD0.COM を図 2.3 として示します。サブルーチンへのパラメータは以下のとおりです。

レジスタ AH ..... ファンクション (0:加算、1:減算、2:乗算、  
3:除算)

レジスタ BX ..... 被加数、被減数、被乗数、被除数

レジスタ CX ..... 加数、減数、乗数、除数

結果は、レジスタ AX (DX, 乗算、除算の場合) に返されます。

### ■図 2.3 INTDO.ASM ソースリスト

```

;
;*****
;
;      INTDO.ASM
;
;      INT ODOHによって加減乗除を実現するサンプル
;
;      このプログラムを実行することによって、INT ODOHを加減乗除を
;      行うソフトウェア割り込みに定義します。
;      このプログラムを実行したのちは、INT ODOHをプログラム中で
;      使用することができます。
;
;      COPYRIGHT(C) 1987 BY SHUWA SYSTEM TRADING CO.,LTD.
;
;      LAST MODIFIED ON FEBRUARY 3RD,1987
;*****
;
CODE      SEGMENT
ASSUME    CS:CODE,DS:CODE,ES:CODE,SS:CODE
;
;      ORG      100H
;
INTDO     PROC
MOV       AH,25H          ; 割り込みベクタのセット
MOV       AL,ODOH         ; 対象の割り込みベクタ
LEA       DX,ENTRY        ; 割り込み処理ルーチンのアドレス
INT       21H
;
LEA       DX,MESSAGE      ; 割り込みベクタに機能を定義した
MOV       AH,9            ; 旨のメッセージを出力
INT       21H
;
MOV       AX,3100H        ; 常駐終了
MOV       DX,100H
INT       21H
;
INTDO     ENDP
;
MESSAGE  DB      'INT ODOHを定義しました。'
          DB      13,10,'$'
;
ENTRY    PROC            ; INT ODOHに対する割り込み処理
;
          PUSH     BX      ; もとの数は保存する
          PUSH     CX
;
          XCHG     AL,AH   ; レジスタAHに応じた処理アドレスを計算

```

```

        XOR     AH,AH
        SHL     AX,1
        MOV     SI,AX
        CALL    CS:ADDR_TABLE[SI]      ; 各処理へ分岐
;
        POP     CX
        POP     BX
        IRET
;
ADDR_TABLE DW     ADDITION      ; 各処理ルーチンのアドレス
           DW     SUBTRACT
           DW     MULTI
           DW     DIVISION
;
ADDITION  PROC     NEAR      ; 加算の処理を行う
        MOV     AX,BX
        ADD     AX,CX
        RET
ADDITION  ENDP
;
SUBTRACT  PROC     NEAR      ; 減算の処理を行う
        MOV     AX,BX
        SUB     AX,CX
        RET
SUBTRACT  ENDP
;
MULTI     PROC     NEAR      ; 乗算の処理を行う
        MOV     AX,BX
        MUL     CX
        RET
MULTI     ENDP
;
DIVISION  PROC     NEAR      ; 除算の処理を行う
        MOV     AX,BX
        XOR     DX,DX
        DIV     CX
        RET
DIVISION  ENDP
;
ENTRY     ENDP
;
CODE      ENDS
;
END       INTDO

```

## ■図 2.3 INTDO.COM ダンプリスト

```

00000000 : B4 25 B0 D0 8D 16 35 01 CD 21 8D 16 1A 01 B4 09 : 59B
00000010 : CD 21 B8 00 31 BA 00 01 CD 21 49 4E 54 20 30 44 : 4FF
00000020 : 30 48 82 F0 92 E8 88 60 82 B5 82 DC 82 B5 82 BD : 95A
00000030 : 81 44 0D 0A 24 53 51 86 C4 32 E4 D1 E0 8B F0 2E : 75E
00000040 : FF 94 47 01 59 5B CF 4F 01 54 01 59 01 5E 01 8B : 547
00000050 : C3 03 C1 C3 8B C3 2B C1 C3 8B C3 F7 E1 C3 8B C3 : A7E
00000060 : 33 D2 F7 F1 C3 : 3B0

```

## ■図 2.3 INTDO.COM 実行例

```

A>INTDO [F5] ----- 定義する
INT ODOHを定義しました。

A>SYMDEB [F5]
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086]

-A [F5] ----- 実験用のプログラムを作る
31C6:0100 MOV AH,0
31C6:0102 MOV BX,2
31C6:0105 MOV CX,3
31C6:0108 INT DO
31C6:010A

-G=100,10A [F5] ----- 加算を実行
AX=0005 BX=0002 CX=0003 DX=0000 SP=CD29 BP=0000 SI=0000 DI=0000
DS=31C6 ES=31C6 SS=31C6 CS=31C6 IP=010A NV UP EI PL NZ NA PO NC
31C6:010A 06 PUSH ES
-E101 [F5] ----- 引き算へ
31C6:0101 00.01 [F5]
-G=100,10A [F5] ----- 実行
AX=FFFF BX=0002 CX=0003 DX=0000 SP=CD29 BP=0000 SI=0002 DI=0000
DS=31C6 ES=31C6 SS=31C6 CS=31C6 IP=010A NV UP EI PL NZ NA PO NC
31C6:010A 06 PUSH ES
-E101 [F5] ----- かけ算へ
31C6:0101 01.02 [F5]
-G=100,10A [F5] ----- 実行
AX=0006 BX=0002 CX=0003 DX=0000 SP=CD29 BP=0000 SI=0004 DI=0000
DS=31C6 ES=31C6 SS=31C6 CS=31C6 IP=010A NV UP EI PL NZ NA PO NC
31C6:010A 06 PUSH ES
-E101 [F5] ----- わり算へ
31C6:0101 02.03 [F5]
-G=100,10A [F5] ----- 実行
AX=0000 BX=0002 CX=0003 DX=0002 SP=CD29 BP=0000 SI=0006 DI=0000
DS=31C6 ES=31C6 SS=31C6 CS=31C6 IP=010A NV UP EI PL NZ NA PO NC
31C6:010A 06 PUSH ES
-Q [F5]

A>

```

## 2.6 ハードウェアを頻繁にアクセス

### ○考え方

INT 命令以上にシステムについての知識を必要とし、かつ周辺 LSI などに対する知識などを必要とするもので、初心者やハードウェアに疎い場合には、特にわずらわしい存在です。

### ○実現方法

INT 命令を用いて実現可能な機能も、独自のプログラムで処理してみる事です。特にディスクアクセスを独自に行ってみると、INT 1BH を検索されても逃れられますし、キメ細かな処理が可能になります。ただし、ディスクアクセスを独自に行うには、FDC (フロッピディスクコントローラ) や DMAC (DMA コントローラ)、割り込みなどの制御をすべて行わなければならない、かなり面倒です。なお DMAC、タイマ、画面関係、キーボード関係も、INT 命令(BIOS)に頼らずに制御できる範囲ですが、同様にその実現は面倒です。

### ○対処方法

ハードウェアをアクセスする手順はかなり複雑ですので、それなりの弱点ともいふべきものが存在します。たとえば、ディスクアクセスを例にとれば、ディスクアクセスを行うと、かなりの頻度で I/O ポートをアクセスし、逆アセンブルした状態から見ると、かなりの量の I/O 命令が並ぶわけです。また、タイミングをとるための NOP 命令も随所に並ぶ可能性があります。ディスクアクセスに用いる I/O ポートというのは有名ですから、INT 命令を捜す代りに、OUT XX,AL などを探されてしまえば、簡単に見つけられてしまいます。そこで、レジスタを用いて間接的に I/O ポートのアドレスを指定してやる方法が出てきます。これは、I/O ポートのアドレスを一時的にレジスタ DX に入れておき、OUT DX, AL などを用いて、間接的に I/O をアクセスするのです。しかしそれでも、この命令を捜されればおしまいです。

また、ディスクアクセスを行うと、割り込みが必ず発生します。割り込みもタイプが決っていますので、その割り込みがロギングされれば、割り込みの待機位置がわかってしまいます。



# 3

## 目立つ命令をかくす

命令の世界でも、目立つものとそうでないものが必ず存在します。目立つというのは目を付けられやすいので、目立たないように姿勢を変えたりかくれたりする必要があります。ここでは、目立つ命令を目立たないようにするテクニックを紹介しましょう。

## 3.1 INT命令をかくす

INT 命令は、各種のハードウェアの制御を簡便に行ってくれるサービスです。なかでも INT 1BH というのは、プロテクトの世界ではあまりにも有名な、ディスクアクセスのための割り込み命令です。基礎編でも紹介したように、プロテクトをソフト面から解析する場合、まず INT 1BH を探すのが定石となっています。実際多くのソフトウェアのプロテクトがこれではずせません。

せっかく高度なプロテクトをかけても、ここから、チェックルーチンを見つけられては元も子もありません。この、やたらと目立つ INT 命令をかくすテクニックを紹介しましょう。

### ■セグメント外CALLを用いる

#### ○考え方

セグメント外 CALL というのは、INT 命令と似たような動作を

する命令です。INT 命令は、戻りアドレスをスタックに積む前に、フラグもスタックに積みます。これに対し、セグメント外 CALL 命令はフラグをスタックに積みません。大きな違いはこれだけです。よって INT 命令をセグメント外 CALL 命令で代用するには、セグメント外 CALL を行う前に、スタックへフラグを積んでおけばよいわけです。細かな違いもありますがここでは問題としません。

### ○実現方法

しかし、INT 命令が自動的にジャンプ先のアドレスを求めるのに対し、セグメント外 CALL 命令は、自らジャンプ先のアドレスを名乗らなければなりません。そこで INT 命令の動作を、もう少し詳しく追ってみます。

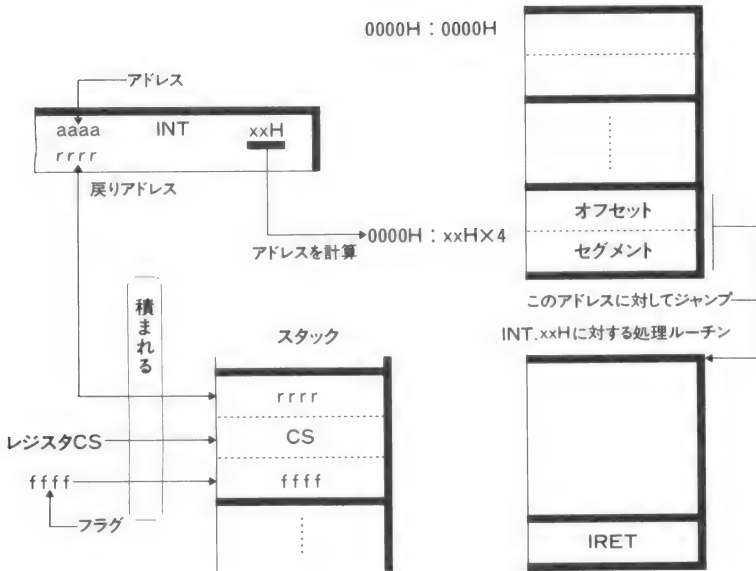
INT 命令が実行されると、フラグと戻りアドレス (CS, IP) が順番にスタックに積まれます。次に割り込みのタイプに対応する割り込みベクタアドレスを算出します。算出は割り込みタイプを 4 倍すれば OK です。これはアドレスのサイズが 4 バイトだからです。割り込みベクタは、セグメント 0000H に配置されていますから、割り込みタイプを 4 倍したアドレスをオフセットとして、そこから格納される 4 バイトのデータをアドレスとみなし、そこへ CALL すればよいのです。この様子を図 3.1 に示しましょう。

これと同じ動作を、セグメント外 CALL 命令で真似ればよいのです。さっそくコーディングしてみましょう。

```
PUSHF      ; フラグのプッシュ
XOR  AX,AX ; 割り込みベクタテーブルのセグメントをセット
MOV  ES,AX
MOV  SI,XX ; ベクタアドレスを計算 (XX は割り込みタイプ)
```



■図 3.1 INT 命令の動作



```
SHL I,1
```

```
SHL I,1
```

```
CALL FAR PTR ES: [SI] ; CALL 命令の実行
```

しかしこれでは多くの命令ステップを必要とし、かつセグメントレジスタを含む多くのレジスタを破壊してしまいます。これを防ぐために、割り込みタイプに対応するベクタアドレスの内容を、プログラム中のデータ領域にあらかじめコピーしておくのです。まず、アドレスの取り出しとコピーは、以下の手順で行えばよいでしょう。

```

XOR  AX,AX; 割り込みベクタテーブルのセグメントをセット
MOV  ES,AX
MOV  SI,XX; ベクタアドレスを計算 (XX は割り込みタイプ)
SHL  SI,1
SHL  SI,1
LES  SI,ES: [SI]; ベクタアドレスの取得
MOV  WORD PTR COPY-VECT,SI;
                                オフセットアドレスのコピー
MOV  WORD PTR COPY-VECT,ES;
                                セグメントアドレスのコピー

```

COPY-VECT は、以下のようにして定義されている変数です。

```
COPY-VECT DD
```

いちどこのようにしておけば、CALL 命令の実行は簡単になります。以下のようにすればよいのです。

```

PUSHF
CALL  COPY-VECT

```

これですと命令ステップも最小で、かつレジスタの破壊を必要とせずに CALL 命令を実行することができるようになります。

### ○対処方法

まず思いつくのは、セグメント外 CALL に対応する命令コードを搜してみることです。しかし、セグメント外 CALL に対応する命令コードというのは、本書で紹介した手順を用いる場合、非常に単純で次のようになっています。

FFH,XX,LO,HI

FFHというのは、セグメント外CALLであることを予感させる命令コードです（このコードで始まる命令には、他にPUSH, POP, INCなどありますので、FFHのみを検索するのは無駄が多すぎます）。次のXXというのはFFHに続く命令コードで、これでセグメント外CALL, PUSH, POP, INCなどの命令を区別します。よって、FFHとこのXXの2バイトを連続して検索してこそ、意味があるわけです。

さてこのXXですが、ジャンプ先のアドレスをメモリに格納しておき、これを直接定数で指定するのであれば1EHに限定することができます。よって、まずはFFH,1EHという並びを搜してみるのも一つの手でしょう。もっともこれでは、ジャンプ先のアドレスを格納してある位置がレジスタによって指定される場合は、まったく効果がありません。アドレスをレジスタによって指定する場合、2バイト目は1EH以外の値になりその値は特定できないからです。

## ■セグメント外RETを用いる

### ○考え方

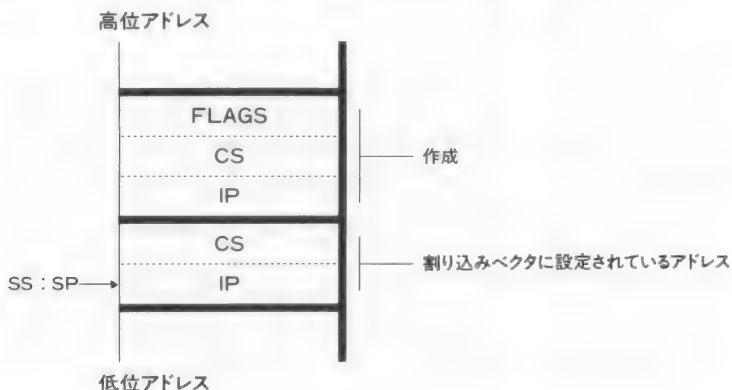
セグメント外CALLと原理は同様ですが、リターン命令でジャンプするところが異なります。リターン命令でジャンプなどできるのかと思われる方は、応用編Ⅰの2を参照してください。

### ○実現方法

割り込みタイプに対応するベクタアドレスの取り出しまでは、セグメント外CALLと同様です。しかし、セグメント外RETを用いた場合、ジャンプ先のアドレスというのは直接指定することがで

きず、スタックに積んでおかねばなりません。またセグメント外 CALL では、自動的にスタックに積まれる INT 命令処理ルーチンから戻るためのアドレスを、スタックに積んでおかねばなりません。セグメント外 RET を行う時点でのスタックのようすは、次の図 3.2 のようになっています。

■図 3.2 セグメント外 RET 実行時のスタックのようす



まず、セグメント外 RET と同様の手順で、割り込みタイプに対応するベクタアドレスの内容をコピーしておきます。そこで、次の手順で命令を実行します。

```

PUSHF      ; フラグのプッシュ
PUSH  CS;  戻りアドレスのスタックへのプッシュ
LEA  AX,RET- ADDR
PUSH  AX
PUSH  WORD PTR COPY- VECT+2
           ; ジャンプ先アドレスのプッシュ
PUSH  WORD PTR COPY- VECT

```

```

DUMMY PROC FAR
    PET
DUMMY ENDP
RET- ADDR:

```

ここで、RET 命令を PROC～ENDP 疑似命令で囲んでいるのは、RET 命令に FAR 属性を持たせるためです。また全体で 5 バイトのデータをスタックに積んでいますが、あとの 2 ワードは RET 命令によって即座に捨てられ、スタックの状態はセグメント外 CALL を行ったときと同じ状態になります。

#### ○対処方法

セグメント外 RET を用いる場合も、セグメント外 RET のコード CBH を搜してみます。しかし、この場合、命令は 1 バイトだけです。ので、非常に効率の悪い検索になってしまいます。

## 3.2 IN/OUT 命令をかくす

IN/OUT 命令もハードウェアと密接し、INT 命令と並んで目立つ命令です。最近では INT 命令によるサービスを用いずに、直接ハードウェアを制御するものも見受けられます。ディスク関係のサービスを行う”INT 1BH”を使わずに、直接 FDC（フロッピディスクコントローラ）、DMAC（DMA コントローラ）などを制御して、プログラムの複雑さを増すなどというのは、その一例です。ここでは、この IN/OUT 命令をかくすテクニックについて紹介しましょう。

### ○考え方

INT 命令のように直接代替となる命令は存在しませんが、工夫次第では、かなり目立たないようにすることも可能です。ここでは間接アドレッシングを用いた方法を示します。

### ○実現方法

レジスタ DX を用いた間接 I/O 指定によって入出力を行います。直接 I/O 指定によって入力を行った場合、命令コードは一元化されやすくなります。

```
EB4 90      IN      AL,90H
```

たとえば、1MB タイプのフロッピーディスクを直接扱う場合には、I/O ポートのアドレスはユーザーズマニュアルなどから明らかですので、このような命令列を捜されるということがあります。


しかし、ここではレジスタ DX による間接指定を用いて、I/O ポートのアドレスをかくすわけです。

```
EC          IN      AL,DX
```

この場合、レジスタ DX に 0090H を設定して命令を実行します。すると、1 命令で上の命令と同じ効果を得ることができます。2 バイトで検索が行われた場合、見つけられる確率は大きくなりますが、1 バイトの場合では見つけられる確率は少なくなります。目くらましのために、コード ECH をプログラム中にばらまいておくのも、解読に対抗する手段としてはよいかも知れません。これは OUT 命令でも同様です。もちろん、解読によってレジスタ DX の内容がわかりにくくなっていなければなりません。

# 4

## MS-DOS版プロテクト技法



プロテクトというのは、正常であるはずのディスクのフォーマットを異常なものにしてしまうのですから、正常なフォーマットを期待する MS-DOS などの OS では、プロテクトのかかっている箇所は読み書きできないのがふつうです。よってプロテクトをかけたならば、その箇所を、OS が使用しないようにしなければなりません。応用編 I の最後では、プロテクトと OS の関係について紹介しましょう。

## 4.1 不良クラスタを作る

### ○考え方

ディスクには、完全というものはありませんから、ときには製品不良や長期間にわたる使用などで、使用できない部分が出てきます。そのような場合の応急処置として、不良クラスタというものを設定して、これを OS が使わないようにする方法があります。

プロテクトをかけられた場所は、OS にとって不良クラスタと同じですから、プロテクトをかけた箇所を不良クラスタに設定すれば、OS はアクセスしなくなります。

ところで MS-DOS では、ファイルをディレクトリと FAT（ファイルアロケーションテーブル）というもので管理しています。

ディレクトリとは、ディスク内に納められるファイルの一覧を記録したものです。ここでは、ディレクトリの構造について特に知る

必要はありません。必要なのは FAT の構造です。

FAT は、ディスク内の使用状況を表にまとめたものとみることができます。FAT の 1.5 バイトが、ディスク上の 1 領域と 1 対 1 で対応しています。FAT を見れば、対応するディスク上の領域がどのように扱われているかを知ることができるのです。まずは、MS-DOS のディスクフォーマットを表 4.1 として示しておきます。

■表 4.1 ディスクフォーマット (MS-DOS)

ディスクタイプ	256KB	1MB	160KB		320KB		640KB	
トラック数	77	154	40	40	80	80	160	160
セクタ数/トラック	26	26	8	9	8	9	8	9
バイト/セクタ	128	1024	512	512	512	512	512	512
予約セクタ数	1	1	1	1	1	1	1	1
FATセクタ数	6	2	1	2	1	2	2	3
FAT数	2	2	2	2	2	2	2	2
ディレクトリセクタ	17	6	4	4	7	7	7	7
セクタ数/クラスタ	4	1	1	1	2	2	2	2
FAT ID	FE	FE	FE	FC	FF	FD	FB	F9

表 4.1 からわかるように MS-DOS では、メディアによって、ディレクトリや FAT の位置や大きさがまちまちです。そこで、MS-DOS ではメディアの判別を容易にするため、FAT ID というシンボルを FAT の先頭においています。この FAT ID を参照することで、メディアの判別を容易にしているのです。

ディスクのフォーマット自体はメディアでまちまちですが、FAT の構造だけは統一されています。基本的には、先ほど説明した 1.5 バイト = 1 クラスタ (2 バイト = 1 クラスタの場合もある。しかし、これはハードディスクなど大容量のメディアの場合なのでここでは無視する) ですが、これが FAT を見るときの障害となっています。まず、どのような状況なのかを説明するために、例をあ



げましょう。図 4.2 に示すのは典型的な FAT の例です（システムディスク品番：PS98-125-HMW）。

■ 図 4.2 典型的な FAT の例（MS-DOS, 1MB）

```
A>symdeb
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086]
-I 0 1 1 ----- アドレス0, ドライブ0(ドライブA), セクタ1, セクタ数1
-d 0
3089:0000 FE FF FF 03 40 00 05 60-00 07 80 00 09 A0 00 0B ~...@...
3089:0010 C0 00 0D E0 00 0F 00 01-11 20 01 13 40 01 15 60 @...@...
3089:0020 01 17 80 01 19 A0 01 18-C0 01 1D E0 01 1F 00 02 .....@...
3089:0030 21 F0 FF 23 40 02 25 60-02 27 80 02 29 A0 02 2B Ip.#@.%'...' ) .+
3089:0040 C0 02 2D E0 02 2F 00 03-31 20 03 33 40 03 35 60 @,~'./,1 ,3@,5'
3089:0050 03 37 80 03 39 A0 03 38-C0 03 3D F0 FF 3F 00 04 ,7.,9 ;@=p,7..
3089:0060 41 20 04 43 40 04 45 60-04 47 80 04 49 A0 04 4B A ,C@,E' ,G.,1 ,K
3089:0070 C0 04 4D E0 04 4F 00 05-51 20 05 53 40 05 55 F0 @.M' ,O.,Q ,S@,Up
-
FAT ID ダミー
```

ここで先頭の 1 バイトは先ほどの FAT ID です。1MB ディスクであることを示すために“FE”の値が書き込まれています。続く 2 バイトの“FF”はダミーです。実体は 4 バイト目から始まるのです。1.5 バイト = 1 クラスタですから、2 クラスタ目からが有効なクラスタ番号ということになります。さて、この 2 クラスタ目に書き込まれている値を取り出すには、図 4.2 で示しているように、4 バイト目に書き込まれている内容と、続く 5 バイト目の内容の下位 4 ビットの値を用います。3 クラスタ目に書き込まれている値を取り出すには、5 バイト目の内容の上位 4 ビットと続く 6 バイト目の内容を用います。このように MS-DOS では、偶数クラスタと奇数クラスタとでは対応する値の取り出し方が異なります。ここで、クラスタ番号を 0 からの値としたバイト位置の算出法の例も兼ねて、値を取り出すための C 言語による関数 `getfat()` を示します。参考にしてください。

```

unsigned getfat(fat, buffer)
unsigned fat;
char * buffer;
{
    int offset;
    if( fat mod 2 ) {
        offset = (fat-1) * 2/3+1;
        return(( buffer [offset] & 0xf0 ) >> 4 +
                buffer [offset+1] << 4 );
    } else {
        offset = fat * 2/3;
        return( buffer [offset+
                butter [offset+1] & 0x0f ) << 8 );
    }
}

```

getfat()の引数である fat と buffer は、それぞれ、クラスタ番号と FAT の読み出されているバッファへのポインタを表しています。

また getfat()自身は、fat に対応するクラスタ自身の情報を返します。

MS-DOS では、クラスタに対応した値によって、そのクラスタの状態を示しています。それらの一覧を表 4.3 にまとめておきます。

実際に FAT を操作するには、SYMDEB などのデバッガを用いることができます。例として FAT ID を書き換えたオペレーションを、図 4.4 として示しておきます。

■表 4.3 FAT の値の意味 (MS-DOS)

値	意 味
0 0 0 H	そのクラスタは未使用
0 0 1 H	使用されない値
0 0 2 H ~ F F 6 H	ファイルとして使用中、続くクラスタの番号を示す
F F 7	不良クラスタ
F F 8 H ~ F F F H	ファイルとして使用中、ファイルの末端を表す

■図 4.4 SYMDEB による FAT 操作

```

A>symdeb
Microsoft Symbolic Debug Utility
Version 3.01
(C) Copyright Microsoft Corp 1984, 1985
Processor is [8086]
-i 0 0 1 1          FAT ID を書き換える
-e 0                00H へ
30B9:0000 FE.0      確認
-d 0                確認
30B9:0000 00 FF FF 03 40 00 05 60-00 07 80 00 09 A0 00 0B .....
30B9:0010 C0 00 0D E0 00 0F 00 01-11 20 01 13 40 01 15 60 .....
30B9:0020 01 17 80 01 19 A0 01 1B-C0 01 1D E0 01 1F 00 02 .....
30B9:0030 21 F0 FF 23 40 02 25 60-02 27 80 02 29 A0 02 2B l p . # % ' . ' . ) +
30B9:0040 C0 02 2D E0 02 2F 00 03-31 20 03 33 40 03 35 60 @ . - . / . 1 . 3 @ . 5 '
30B9:0050 03 37 80 03 39 A0 03 3B-C0 03 3D F0 FF 3F 00 04 . 7 . . 9 . ; @ . = p . 7 .
30B9:0060 41 20 04 43 40 04 45 60-04 47 80 04 49 A0 04 4B A . C @ . E ' . G . . I . K
30B9:0070 C0 04 4D E0 04 4F 00 05-51 20 05 53 40 05 55 F0 @ . M ' . O . . Q . S @ . U p
-w 0 0 1 1        書き戻す
-q

A>chkdsk a:        ディスクを調べる
ディスク MSDOS3_1 は 1986-12-17 15:23 に作成されました
このディスクは扱えません      FAT ID を破壊したのでこうなる
続行しますか <Y/N>? n

A>

```

しかし、この方法では FAT の読み書きや修正を手動で行うため、かなりの危険を伴います。そこで、FAT の書き換えを行うユーティリティ FAT.COM を紹介します。これはユーザの指定するドライブの、指定するトラックに対応する全クラスタの値を、“FF7” (不良クラスタ) へ書き換えるものです。

■図 4.5 FAT.ASM ソースリスト

```

;
;*****
;
;      FAT.ASM
;
;      指定されたトラックに含まれるクラスタを、使用不可とする。
;
;      このプログラムは、以下のフォーマットを持つディスクに
;      対応しています。
;
;      1MBディスク:   バイト/セクタ = 1024
;                      セクタ/トラック = 8
;      640KBディスク: バイト/セクタ = 512
;                      セクタ/トラック = 8
;
;      起動は、パラメータを指定せずに行います。入力は、プログラムの
;      指示に従って下さい。
;
;      COPYRIGHT(C) 1987 BY SHUWA SYSTEM TRADING CO.,LTD.
;
;      LAST MODIFIED ON FEBRUARY 4TH,1987
;
;*****
;
;DRIVE  =      1      ; ドライブ B を操作する (変更可)
;
;CODE   SEGMENT
;ASSUME CS:CODE,DS:CODE,ES:CODE,SS:CODE
;
;      ORG      100H
;
;FAT     PROC
;LEA     DX,OPENING_MSG ; 開始メッセージ表示
;MOV     AH,9
;INT     21H
;
;      PUSH     DS
;MOV     AH,1CH          ; ドライブのタイプを判別する
;MOV     DL,DRIVE+1
;INT     21H
;PUSH     DS
;POP      ES
;POP      DS
;CMP     AL,0FFH         ; 不正なドライブか?
;JNE     GOOD_DRIVE     ; 正しいドライブなら次へ
;
;BAD_DRIVE:
;      LEA     DX,BAD_DRIVE_MES; ドライブが不正である旨のメッセージを出力
;MOV     AH,9
;INT     21H
;JMP     EXIT
;
;      GOOD_DRIVE:
;CMP     CX,1024         ; 1MBディスクか?
;JE      MAIN
;
;      CMP     CX,512         ; 640KBディスクか?
;JNE     BAD_DRIVE      ; それ以外のディスクではエラー
;
;      CMP     ES:BYTE PTR [BX],0FBH ; セクタ/トラック=8か?
;JNE     BAD_DRIVE
;
;      MAIN:
;MOV     ALLOCSIZE,AL    ; セクタ/クラスタをセット
;MOV     MAX_CLUSTER,DX ; クラスタ数をセット
;
;

```

```

LEA    DX,PROMPT_TRACK ; 使用不可にするトラックを入力
MOV     AH,9
INT     21H
LEA     DX,IN_BUFFER
MOV     AH,10
INT     21H
LEA     BX,IN_BUFFER+2 ; 入力数値をバイナリへ変換
XOR     DL,DL
;
GET_NUM_LOOP:
MOV     AL,[BX] ; 数値文字データを取り出す
CMP     AL,'0'
JB      CHECK_TRACK ; 数字でないなら変換終了
;
CMP     AL,'9'
JA      CHECK_TRACK ; 数字でないなら変換終了
;
SUB     AL,'0' ; ASCII文字をバイナリへ変換
MOV     DH,DL ; レジスタDLを10倍する
SHL     DL,1
SHL     DL,1
ADD     DL,DH
SHL     DL,1
ADD     DL,AL ; 新たに加える
INC     BX ; 次の桁へ
JMP     GET_NUM_LOOP
;
CHECK_TRACK:
MOV     AL,DL ; トラック番号をセット
MOV     TRACK,AL
XOR     AH,AH ; レコード番号へ変換
SHL     AX,1
SHL     AX,1
SHL     AX,1
CMP     ALLOCSIZE,1 ; 1MBか640KBディスクか調べる
JNE     CHECK_640KB ; 640KBディスクとしてチェック
;
CHECK_1MB: ; 1MBディスクとしてチェック
CMP     AX,11 ; 予約域であるかチェック
JB      BAD_TRACK
;
SUB     AX,9 ; レコード番号をクラスタ番号へ変換
CMP     AX,MAX_CLUSTER ; 最大クラスタ番号を越えていないかチェック
JB      GET_FAT ; 越えていなければFAT読み出しへ
;
BAD_TRACK:
LEA     DX,BAD_TRACK_MSG1
MOV     AH,9
INT     21H
JMP     EXIT
;
CHECK_640KB: ; 640KBディスクとしてチェック
CMP     AX,12 ; 予約域であるかチェック
JB      BAD_TRACK
;
SUB     AX,10 ; レコード番号をクラスタ番号へ変換
SHR     AX,1
ADD     AX,2
CMP     AX,MAX_CLUSTER ; 最大クラスタ番号を越えていないかチェック
JNB     BAD_TRACK ; 越えていればエラーを出力
;
GET_FAT: ; FAT読み出しを行う
MOV     CLUSTER,AX ; クラスタ番号をセット
MOV     AL,DRIVE ; 読み出しドライブ
LEA     BX,FAT_BUFFER ; 読み出しバッファ
MOV     CX,2 ; 読み出しレコード数 (FATセクタ数)
MOV     DX,1 ; 読み出し開始レコード番号 (FAT開始セクタ)

```

```

INT      25H
POP      AX
JNC      CHECK_FAT      ; INT 25Hのためのダミーポップ
;
;
;      LEA      DX,READ_ERROR_MSG      ; 読み出しエラーメッセージを出力
MOV      AH,9
INT      21H
JMP      EXIT

;
CHECK_FAT:
MOV      AX,CLUSTER      ; トラックが使用されているかチェック
MOV      BP,AX      ; クラスタ番号を取り出す
MOV      AL,ALLOC_SIZE      ; 1トラックを占めるクラスタ数を算出
XOR      AH,AH
XOR      AX,11B      ; 結果は1MBディスクなら8, 640KBディスクなら4
SHL      AX,1
SHL      AX,1
MOV      CX,AX

;
CHECK_FAT_LOOP:
MOV      AX,BP
TEST     AX,1      ; 奇数クラスタか?
JNZ      ODD_CLUSTER      ; 奇数クラスタの処理を行う

;
EVEN_CLUSTER:
MOV      SI,3      ; 偶数クラスタの処理を行う
MUL      SI      ; クラスタに対応するバッファ内の位置を算出
SHR      AX,1
MOV      SI,AX

;
MOV      AX,FAT_BUFFER[SI]; クラスタに対応する値を取り出す
AND      AX,0FFFH      ; 上位4ビットを無効とする
JMP      CHECK_VALUE      ; FAT値の調査へ

;
ODD_CLUSTER:
DEC      AX      ; 奇数クラスタの処理を行う
MOV      SI,3      ; クラスタに対応するバッファ内の位置を算出
MUL      SI
SHR      AX,1
INC      AX
MOV      SI,AX

;
MOV      AX,FAT_BUFFER[SI]; クラスタに対応する値を取り出す
SHR      AX,1      ; 下位4ビットを無効とする
SHR      AX,1
SHR      AX,1

;
CHECK_VALUE:
OR      AX,AX      ; FAT値の調査
JZ      NEXT_CLUSTER      ; 未使用か?
;
;      LEA      DX,BAD_TRACK_MSG2
MOV      AH,9
INT      21H
JMP      EXIT

;
NEXT_CLUSTER:
INC      BP      ; 次のクラスタへ
LOOP     CHECK_FAT_LOOP      ; 1トラックをチェックする

;
SET_FAT:
MOV      AX,CLUSTER      ; トラックに含まれるクラスタを使用不可へ
MOV      BP,AX      ; クラスタ番号を取り出す
MOV      AL,ALLOC_SIZE      ; 1トラックを占めるクラスタ数を算出
XOR      AH,AH
XOR      AX,11B      ; 結果は1MBディスクなら8, 640KBディスクなら4

```

```

        SHL     AX,1
        SHL     AX,1
        MOV     CX,AX
;
SET_FAT_LOOP:
        MOV     AX,BP
        TEST    AX,1          ; 奇数クラスタか?
        JNZ     ODD_CLUSTER1  ; 奇数クラスタの処理を行う
;
EVEN_CLUSTER1:
        MOV     SI,3          ; 偶数クラスタの処理を行う
                                ; クラスタに対応するバッファ内の位置を算出
        MUL     SI
        SHR     AX,1
        MOV     SI,AX
;
        MOV     AX,FAT_BUFFER[SI] ; クラスタに対応する値を取り出す
        AND     AX,0F00H        ; 上位4ビットを残して無効とする
        OR      AX,0FF7H        ; 使用不可クラスタ情報をセット
        JMP     SET_VALUE       ; FAT値のセット
;
ODD_CLUSTER1:
        DEC     AX             ; 奇数クラスタの処理を行う
                                ; クラスタに対応するバッファ内の位置を算出
        MOV     SI,3
        MUL     SI
        SHR     AX,1
        INC     AX
        MOV     SI,AX
;
        MOV     AX,FAT_BUFFER[SI] ; クラスタに対応する値を取り出す
        AND     AX,0FH          ; 下位4ビットを残して無効とする
        OR      AX,0FF70H        ; 使用不可クラスタ情報をセット
;
SET_VALUE:
                                ; FAT値のセット
        MOV     FAT_BUFFER[SI],AX
        INC     BP              ; 次のクラスタへ
        LOOP    SET_FAT_LOOP    ; 1トラックをセットする
;
GET_SURE:
        LEA     DX,PROMPT_SURE  ; FAT書き込み確認のメッセージを出力
        MOV     AH,9
        INT     21H
        LEA     DX,IN_BUFFER    ; 応答を入力
        MOV     AH,10
        INT     21H
;
        MOV     AL,IN_BUFFER+2  ; バッファから取り出す
        AND     AL,0DFH         ; 大文字へ変換
        CMP     AL,'Y'
        JE      PUT_FAT         ; YESならばFATを書き戻す
;
        CMP     AL,'N'
        JNE     GET_SURE
;
PUT_FAT:
                                ; FATを書き戻す
        MOV     AL,DRIVE        ; 書き込みドライブ
        LEA     BX,FAT_BUFFER   ; 書き込みバッファ
        MOV     CX,2            ; 書き込みレコード数 (FATセクタ数)
        MOV     DX,1            ; 書き込み開始レコード番号 (FAT開始セクタ)
        INT     26H
        POP     AX              ; INT 26Hのためのダミーポップ
        JNC     EXIT            ; 書き込みエラーが発生していなければ終了
;
        LEA     DX,WRITE_ERROR_MSG ; 書き込みエラーメッセージを出力
        MOV     AH,9
        INT     21H
;
EXIT:

```

```

MOV     AX,4C00H      ; 非常駐終了
INT     21H

;
; ALLOCSIZE           DB    ?      ; セクタ/クラスタ (1MBディスクでは1,
;                                     640KBディスクでは2)
; MAX_CLUSTER         DW    ?      ; クラスタ数 (最大クラスタ番号 + 1)
;
; TRACK              DB    ?      ; トラック
;
; CLUSTER             DW    ?      ; クラスタ番号
;
; IN_BUFFER           DB    10,?   ; キー入力用バッファ
;                   DB    10 DUP(?)
;
; OPENING_MSG         DB    13,10
;                   DB    'ドライブ B: の指定トラックを使用不可にします。'
;                   DB    13,10,'$'
;
; BAD_DRIVE_MES       DB    'ドライブは操作できません。'
;                   DB    13,10,'$'
;
; PROMPT_TRACK        DB    13,10
;                   DB    '使用不可にするトラックを入力して下さい:'
;                   DB    '$'
;
; BAD_TRACK_MSG1      DB    13,10
;                   DB    '指定トラックはファイル領域ではありません。'
;                   DB    13,10,7,'$'
;
; BAD_TRACK_MSG2      DB    13,10
;                   DB    '指定トラックはファイルに割り当てられています。'
;                   DB    13,10,7,'$'
;
; READ_ERROR_MSG      DB    13,10
;                   DB    'FATを読み出せません。'
;                   DB    13,10,7,'$'
;
; WRITE_ERROR_MSG     DB    13,10
;                   DB    'FATを書き戻せません。'
;                   DB    13,10,7,'$'
;
; PROMPT_SURE         DB    13,10
;                   DB    '指定トラックを使用不可にします。'
;                   DB    13,10
;                   DB    'よろしいですか(Y/N)? '
;                   DB    '$'
;
; FAT_BUFFER          DW    1024 DUP(?) ; FAT読み出し用バッファ
;
; FAT                 ENDP
;
; CODE               ENDS
;
;                   END          FAT

```

■図 4.5 FAT.COM ダンプリスト

```

00000000 : 8D 16 B9 02 B4 09 CD 21 1E B4 1C B2 02 CD 21 1E : 587
00000010 : 07 1F 3C FF 75 08 8D 16 EC 02 B4 09 CD 21 E9 81 : 687
00000020 : 01 81 F9 00 04 74 0C 81 F9 00 02 75 E9 26 80 3F : 58E
00000030 : F8 75 E3 A2 A7 02 89 16 A8 02 8D 16 09 03 B4 09 : 653
00000040 : CD 21 8D 16 AD 02 B4 0A CD 21 8D 1E AF 02 32 D2 : 64C
00000050 : 8A 07 3C 30 72 15 3C 39 77 11 2C 30 8A F2 D0 E2 : 60B
00000060 : D0 E2 02 D6 D0 E2 02 D0 43 EB E5 8A C2 A2 AA 02 : 9BB

```



```

00000070 : 32 E4 D1 E0 D1 E0 D1 E0 80 3E A7 02 01 75 19 3D : 85C
00000080 : 08 00 72 09 2D 09 00 38 06 A8 02 72 1E 8D 16 34 : 30E
00000090 : 03 84 09 CD 21 E9 0A 01 3D 0C 00 72 F0 2D 0A 00 : 484
000000A0 : D1 E8 05 02 00 38 06 A8 02 73 E2 A3 A8 02 80 01 : 601
000000B0 : 8D 1E 08 04 B9 02 00 BA 01 00 CD 25 58 73 08 8D : 482
000000C0 : 16 98 03 84 09 CD 21 E9 D8 00 A1 A8 02 88 E8 A0 : 77E
000000D0 : A7 02 32 E4 35 03 00 D1 E0 D1 E0 88 C8 88 C5 A9 : 8A5
000000E0 : 01 00 75 13 8E 03 00 F7 E6 D1 E8 88 F0 88 84 08 : 772
000000F0 : 04 25 FF 0F E8 18 90 48 8E 03 00 F7 E6 D1 E8 40 : 7A9
00000100 : 8B F0 88 84 08 04 D1 E8 D1 E8 D1 E8 08 C0 : A45
00000110 : 74 08 8D 16 64 03 84 09 CD 21 E9 85 00 45 E2 8D : 686
00000120 : A1 A8 02 88 E8 A0 A7 02 32 E4 35 03 00 D1 E0 D1 : 7DA
00000130 : E0 88 C8 88 C5 A9 01 00 75 16 BE 03 00 F7 E6 D1 : 827
00000140 : E8 88 F0 88 84 08 04 25 00 F0 0D F7 0F E8 16 90 : 737
00000150 : 48 BE 03 00 F7 E6 D1 E8 40 88 F0 88 84 08 04 25 : 79A
00000160 : 0F 00 0D 70 FF 89 84 08 04 45 E2 C7 8D 16 CE 03 : 606
00000170 : 84 09 CD 21 8D 16 AD 02 84 0A CD 21 A0 AF 02 24 : 61E
00000180 : DF 3C 59 74 04 3C 4E 75 E3 80 01 8D 1E 08 04 89 : 5EF
00000190 : 02 00 8A 01 00 CD 26 58 73 08 8D 16 83 03 84 09 : 499
000001A0 : CD 21 88 00 4C CD 21 00 00 00 00 00 00 0A 00 00 : 2EA
000001B0 : 00 00 00 00 00 00 00 00 00 00 0A 83 68 83 89 83 : 291
000001C0 : 43 83 75 20 42 3A 20 82 CC 8E 77 92 E8 83 67 83 : 731
000001D0 : 89 83 62 83 4E 82 F0 8E 67 97 70 95 73 89 C2 82 : 882
000001E0 : C9 82 85 82 DC 82 87 81 44 0D 0A 24 83 68 83 89 : 78E
000001F0 : 83 43 83 75 82 CD 91 80 8D EC 82 C5 82 AB 82 DC : 969
00000200 : 82 89 82 F1 81 44 0D 0A 24 0D 0A 8E 67 97 70 95 : 656
00000210 : 73 89 C2 82 C9 82 87 82 E9 83 67 83 89 83 62 83 : 908
00000220 : 4E 82 F0 93 FC 97 CD 82 B5 82 C4 89 8A 82 B3 82 : A2A
00000230 : A2 3A 20 24 0D 0A 8E 77 92 E8 83 67 83 89 83 62 : 691
00000240 : 83 4E 82 CD 83 74 83 40 83 43 83 88 97 CC 88 E6 : 87F
00000250 : 82 C5 82 CD 82 A0 82 E8 82 DC 82 89 82 F1 81 44 : 9F3
00000260 : 0D 0A 07 24 0D 0A 8E 77 92 E8 83 67 83 89 83 62 : 583
00000270 : 83 4E 82 CD 83 74 83 40 83 43 83 88 82 C9 8A 84 : 807
00000280 : 82 E8 93 96 82 C4 82 E7 82 EA 82 C4 82 A2 82 DC : A76
00000290 : 82 87 81 44 0D 0A 07 24 0D 0A 46 41 54 82 F0 93 : 537
000002A0 : C7 82 DD 8F 6F 82 89 82 DC 82 89 82 F1 81 44 0D : 93D
000002B0 : 0A 07 24 0D 0A 46 41 54 82 F0 8F 91 82 A8 96 DF : 65B
000002C0 : 82 89 82 DC 82 89 82 F1 81 44 0D 0A 07 24 0D 0A : 665
000002D0 : 8E 77 92 E8 83 67 83 89 83 62 83 4E 82 F0 8E 67 : 892
000002E0 : 97 70 95 73 89 C2 82 C9 82 B5 82 DC 82 87 81 44 : 938
000002F0 : 0D 0A 82 E6 82 E8 82 B5 82 A2 82 C5 82 87 82 A9 : 8F2
00000300 : 28 59 2F 4E 29 3F 20 24 00 00 00 00 00 00 00 : 1AA

```

FAT.COM はパラメータを与えずに実行します。あとは、FAT.COM の指示に従ってパラメータを入力してください。FAT.COM は、指定されたトラックに含まれるクラスタのうち、1個でもすでに使用されている場合には、エラーを出力して動作を停止します。

FAT.COM の使用例を示しましょう。ここではシステムディスク (PS98-125-HMW) において使用されていない最終トラックに、セクタ長を変化させる (1セクタ=256バイト) というプロテクトをかけるものです。

## ■図 4.6 最終トラックにプロテクトをかける例

```

A>CHKDSK B: [Enter]          まずディスクをチェックしておく

1250304 バイト : 全ディスク容量
61440   バイト : 2 個のシステムファイル
1154048 バイト : 47 個のユーザーファイル
34816   バイト : 使用可能ディスク容量

655360 バイト : 全メモリ
497664 バイト : 使用可能メモリ

A>FAT [Enter]                FAT.COMを実行

ドライブ B: の指定トラックを使用不可にします。

使用不可にするトラックを入力して下さい: 153 ——トラックを入力
指定トラックを使用不可にします。
よろしいですか (Y/N)? Y [Enter]
A>CHKDSK B: [Enter]          再びディスクをチェック

1250304 バイト : 全ディスク容量
61440   バイト : 2 個のシステムファイル
1154048 バイト : 47 個のユーザーファイル
8192    バイト : 不良セクタ ——不良セクタが確認できる
26624   バイト : 使用可能ディスク容量

655360 バイト : 全メモリ
497664 バイト : 使用可能メモリ

A>SYMDEB [Enter]            プロテクトを施す
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086]
-A [Enter]                  プログラムを入力
2F6D:0100 MOV AH,5D
2F6D:0102 MOV AL,91
2F6D:0104 MOV BX,4
2F6D:0107 MOV CH,1
2F6D:0109 MOV CL,4C ——セクタ長1 (=256/バイト)のプロテクトをかける
2F6D:010B MOV DH,1
2F6D:010D MOV DL,FF
2F6D:010F MOV BP,200
2F6D:0112 INT 1B
2F6D:0114
-E 200 4C 01 01 01 [Enter]  IDをセット
-G=100,114 [Enter]          実行
AX=0091 BX=0004 CX=014C DX=01FF SP=CF82 BP=0200 SI=0000 DI=0000
DS=2F6D ES=2F6D SS=2F6D CS=2F6D IP=0114 NV UP EI PL NZ NA PO NC
2F6D:0114 06 PUSH ES
-Q [Enter]

A>DISKCOPY A: B: /V [Enter] ——もとのディスクと比較
DISKCOPY version 2.1

ディスクの照会を行います

送り側ディスクをドライブ A: に挿入してください
受け側ディスクをドライブ B: に挿入してください
準備ができたならどれかのキーを押してください

トラック 0 の内容が異っています
中止 <A> , 強行 <I> ? I [Enter] ——FATが異なっている

トラック 153 で読み込みを失敗しました
中止 <A> , 再試行 <R> , 強行 <I> ? A [Enter] ——プロテクトがかかっている

```

---

照合は失敗しました  
別のディスクを照合しますか <Y/N>? N 

---

## 4.2 ダミーファイルを作る

4.1 では、FAT を操作して使用不可能なクラスタを設け、そこにプロテクトをかけるという方法を説明しましたが、次に、FAT を操作するという点では同じで、さらにディレクトリを操作してダミーファイルを作るという方法を説明しましょう。

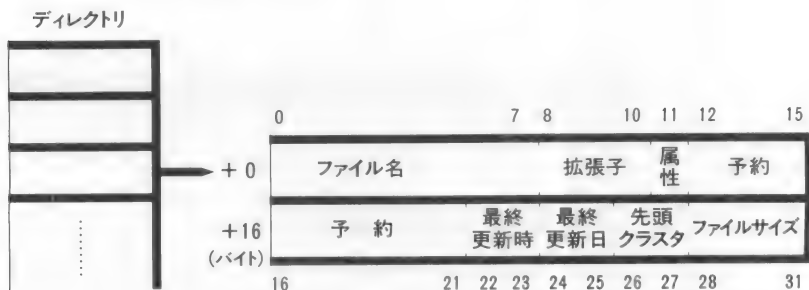
不良クラスタを設ける方法では FAT 内を眺めると、不自然なことがはっきりとわかってしまって、プロテクト向きではありません。そこで、ダミーファイルを設け、ファイルの一部にプロテクトをかけてしまうのです。もちろんそのファイルは、OS から読み書きすることはできません。極端な話、あらゆるアクセスを禁止しなければなりません（アクセスさせ、エラーの発生を確認するという逆説的使用法もあります）。

ダミーファイルにプロテクトをかけるには、すでに存在するファイルにプロテクトをかけるほうが楽でもあるし自然です。なぜなら、MS-DOS が新たにファイルを作るとき、作られる位置は外からはわからないからです。また、FAT を操作して、強引にプロテクトをかけたトラックにファイルの位置を持ってきても、何となく不自然です。そこで、すでに存在するファイルの位置を捜して、そこにプロテクトをかけてしまうのです。

ファイルの位置を見つけるには、FAT のほかにディレクトリも参照します。ディレクトリの位置については、4.1 を参照してくだ

さい。ここではディレクトリの構造について説明します。

■図 4.7 MS-DOS のディレクトリ



MS-DOS のディレクトリは、1 個のファイルについて 32 バイトで構成され、そこにファイル名や属性、サイズなどの情報が格納されています。図 4.7 からわかるように、ディレクトリから目的のファイル名を捜し、そのファイルに対応する先頭のクラスタを見つけ、そのクラスタを含むトラックに対してプロテクトをかければよいのです。

## 4.3 チェックの方法

さて、はじめにも触れたようにプロテクトというものは、それをチェックして初めてその効果が現れます。初期のプロテクトと異なり、現在ではコピープログラムを停止させるのが目的ではないので、きちんとコピーされているかどうかをチェックすることは、どうしても必要なわけです。

チェックを行うにはいくつかの方法があります。まずプログラムの先頭においてチェックを行い、チェックに失敗したら、そこでプログラムを終了させてしまうものです。しかしこれはプログラムの先頭でチェックを済ませてしまうため、解読によってチェックルーチンがを見つけられやすいという欠点を持ちます。この方法は、現在では姿を消す傾向にあり、次に示すようなチェックルーチンをばらまくという方法に移っています。

プログラムの要所要所でチェックを行い、チェックに成功したことを確認してから、実行を継続させるというものです。最初はうまく動いていても、途中で動かなくなる可能性があるわけで、チェックルーチンが見つかりにくく、現在ではこの方法が主流になりつつあります。

しかし、プロテクトモジュールを外部で作成している場合、それを組み込む都合上、どうしてもプログラムの先頭にチェックルーチンを置く必要が出てくる場合があります。また、チェックルーチンによるディスクアクセスによって、本当に必要であるディスク操作が遅くなるなどの支障の出る場合もあります。

## 応用編 II





1. ツールに対抗する
2. 暗号化のテクニック
3. プログラムをかくす
4. 錯乱のためのテクニック
5. ワナをかける
6. 既存の知識を破棄させる
7. プログラム実行のテクニック

---

応用編 I では、プロテクトをかける側に立った手法として、比較的初歩的なものを扱ってきました。続けて応用編 II では、マシンや CPU の知識を限りなく活かす、かなり高度ともいえるテクニックを紹介していきます。

## 1

## ツールに対抗する

DEBUG, SYMDEBをはじめとするデバッガは、プログラム解析の際によく用いられるツールですが、これにはそれなりの弱点があります。なぜなら、これらは MS-DOS の環境に合わせて動作するプログラムのためにあるので、ふだんプログラム解析に用いられるようなものではないからです。現在、多くのユーザがこれらを用いて解析しているとすれば、それを封じることは簡単です。

ここでは、この SYMDEBをはじめとする、解析ツールの機能に対抗するようなテクニックを集めて紹介します。

なお、いずれもプログラムが実行された場合にのみ効果を持つものですので、プログラムを実行させない、すなわち逆アセンブルリストの解読に対抗するには、応用編 I に紹介したテクニックを用い、できるだけプログラムを実行させる手段に出る必要があります。

## 1.1 ツールの命を無効にする

### ■INT3命令を無効にする

#### ○考え方

ブレークポイントを置いてプログラムの実行を追っている場合に効果があります。ふつう、ブレークポイントを置くということは、INT3 命令を置くことなのです。INT3 命令は INT 命令の特殊なものであり、1 バイトで 1 命令を構成することができます。INT 命令は 2 バイトで構成されますから、ブレークポイントとして適しているのです（もっとも、この目的で用意された命令なのですが）。



## ○実現方法

SYMDEB では、次のようにプログラムの実行コマンドが入力されると、以下のようなことを行います。

-g = 100,13f 

まず実行開始アドレス（待避してあるレジスタ IP の内容）を 0100H に設定し、ワークエリアに待避してあるレジスタの内容（R コマンドで表示されるもの）を実際のレジスタの内容とし、実行終了アドレス（013FH）に INT3 命令を置いてから、実行開始アドレスへジャンプします。あとは、メモリ上にある指定されたアドレスからのプログラムが実行されるわけですが、実行がブレークポイントの設定されている 013FH にくると、INT3 命令が実行されます。

INT3 命令が実行されると、INT 1BH などと同様、対応する割り込みベクタに格納されているアドレスにジャンプしますが、ここにはブレークポイント処理ルーチンとして、レジスタの内容をワークエリアにセーブし実行をデバッガに戻すルーチンを、デバッガ自身が置いています。要するに、INT3 に対する処理ルーチンはデバッガが置いているのですから、INT3 が実行される前に解析されるプログラムのほうで、INT3 割り込みを封じてしまえばよいのです。INT3 が入ったとたんプログラムを暴走させてしまうのが、もっとも単純ともいえる方法でしょう。

INT3 に対する処理ルーチンは、解析されるプログラムが自分で持っています。プログラムの冒頭で INT3 のベクタをこの新しいルーチンのアドレスに書き換えれば OK です。

INT3 を封じるには、プログラムを書き換えながら実行するという手もあります。すなわちブレークポイントを設定するような場所

を書き換えながら実行すれば、ブレークポイントは抹消され、実行は停止されないわけです。

### ○対処法

INT3 に対応する割り込みベクタを書き換える部分を、実行させないようにするのが一番です。とりあえずプログラムを走らせて、まともに動作しなくなる区間を絞ります。

また、INT3 に対応する割り込みベクタを書き換えるという手段がありますが、プログラムの実行中に書き換えられてしまうのでは手の打ちようがありません。しかし、多少手のこんだ方法ですが、シングルステップ割り込みを用いるという方法があります。

8086 は、トレースフラグ (TF) が 1 であるときには、1 命令実行ごとに INT 01H と等価な割り込みを発生させます。通常、この割り込みはプログラムのトレースに用いられているのですが、ブレークポイントを設定している場合には、とりあえずトレースは行う必要がないのですから、ここで、シングルステップ割り込みを INT3 ベクタの復帰に用いるのです。するとプログラムの実行において、1 命令ごとに INT 01H の割り込みが入りますが、ここで常に INT3 に対するベクタを書き改めるのです。そうすれば、たとえある命令において割り込みベクタが書き換えられても、次の命令を実行する時点においては、割り込みベクタは元の状態に戻っているわけです。

ただし、この対処法は完璧ではありません。後述するようにシングルステップ割り込みを封じられたら、ひとたまりもありません。別の手段として、インターバルタイマを用いて、割り込みベクタを定期的書き改めるというのもありますが、インターバルタイマを封じられたら使用できませんし、また、シングルステップ割り込みを行うほどのキメ細かな操作もできません。また、SYMDEB にこ

のような機能を付加するのは困難です。新たに解析用プログラムを作成するという場合の、参考程度にとどめておけばよいでしょう。

## ○サンプル

INT3 に対応する割り込みベクタを、プログラムの先頭で書き換えて、INT3 に会おうと、即座に SYMDEB を終了してしまうサンプルプログラム INT3.COM を、図 1.1 として示します。実行例も SYMDEB 上で動作させて示してあります。

■図 1.1 INT3.ASM ソースリスト

```

;
;*****
;
;      INT3.ASM
;
;      SYMDEB の機能を無効にするサンプル ( 1 )
;
;      このプログラムを実行することによって、INT3 による
;      ブレークポイントの機能を無効にできます。
;
;      COPYRIGHT(C) 1987 BY SHUWA SYSTEM TRADING CO.,LTD.
;
;      LAST MODIFIED ON FEBRUARY 4TH,1987
;
;*****
CODE    SEGMENT
        ASSUME    CS:CODE,DS:CODE
;
;      ORG        100H
;
;_INT3   PROC
;
;      LEA        DX,MESSAGE      ; 警告の表示
;      MOV        AH,9
;      INT        21H
;
;      MOV        AX,2503H        ; 割り込みベクタの設定
;      LEA        DX,ENTRY
;      INT        21H
;
;      LEA        DX,ENDING        ; おじに終了したことを告げるメッセージ
;      MOV        AH,9
;      INT        21H
;
;      MOV        AX,4C00H
;      INT        21H
;
;_ENTRY  PROC      FAR
;
;      LEA        DX,ESCAPE        ; 終了メッセージの表示
;      MOV        AH,9

```

```

INT     21H
;
MOV     ES,DS:[0AH]      ; プログラム終了アドレスを取得
LES     AX,ES:[0AH]      ; 親のプログラム終了アドレスを取得
MOV     DS:WORD PTR [0AH],AX ; プログラム終了アドレスをコピー
MOV     DS:WORD PTR [0CH],ES
;
MOV     AX,4C01H         ; プログラム終了 (int trap haltが発生)
INT     21H
;
ENTRY   ENDP
;
MESSAGE DB      13,10
DB      'ここから先は危険ですよ!!'
DB      13,10,'$'
;
ENDING  DB      13,10
DB      'ぶじプログラムは実行されました。'
DB      13,10,'$'
;
ESCAPE  DB      13,10
DB      'さようなら。。。SYMDEB。。。'
DB      13,10,'$'
;
_INT3   ENDP
;
CODE    ENDS
;
END     _INT3

```

## ■図 1.1 INT3.COM ダンプリスト

```

00000000 : 8D 16 3B 01 B4 09 CD 21 B8 03 25 8D 16 1E 01 CD : 4F9
00000010 : 21 8D 16 5A 01 B4 09 CD 21 B8 00 4C CD 21 8D 16 : 55F
00000020 : 7F 01 B4 09 CD 21 8E 06 0A 00 26 C4 06 0A 00 A3 : 466
00000030 : 0A 00 8C 06 0C 00 B8 01 4C CD 21 0D 0A 82 B1 82 : 467
00000040 : B1 82 A9 82 E7 90 E6 82 CD 8A EB 8C AF 82 C5 82 : A83
00000050 : 87 82 E6 81 49 81 49 0D 0A 24 0D 0A 82 D4 82 B6 : 693
00000060 : 83 76 83 8D 83 4F 83 89 83 80 82 CD 8E C0 8D 73 : 887
00000070 : 82 83 82 EA 82 DC 82 B5 82 BD 81 44 0D 0A 24 0D : 782
00000080 : 0A 82 B3 82 E6 82 A4 82 C8 82 E7 81 44 81 44 81 : 88B
00000090 : 44 81 43 53 59 4D 44 45 42 81 44 81 44 81 44 0D : 528
000000A0 : 0A 24 : 02E

```

## ■図 1.1 INT3.COM 実行例

```

A>INT3 [F5] —————ダイレクトに実行

ここから先は危険ですよ!!

ぶじプログラムは実行されました。——実行された

A>SYMDEB B:\CMDS\INT3.COM [F5] —————デバグで動作させてみる
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086]

```

```

-U [F5] ----- 確認のため逆アセンブル
30C1:0100 8D163B01    LEA    DX,[013B]
30C1:0104 B409       MOV    AH,09
30C1:0106 CD21       INT    21
30C1:0108 B80325     MOV    AX,2503
30C1:010B 8D161E01    LEA    DX,[011E]
30C1:010F CD21       INT    21
30C1:0111 8D165A01    LEA    DX,[015A]
30C1:0115 B409       MOV    AH,09
-G=100,108 [F5] ----- 最初のメッセージを表示させてみる

ここから先は危険ですよ!!
AX=0924 BX=0000 CX=00A2 DX=013B SP=FFFE BP=0000 SI=0000 DI=0000
DS=30C1 ES=30C1 SS=30C1 CS=30C1 IP=0108 NV UP EI PL NZ NA PO NC
30C1:0108 B80325     MOV    AX,2503
-U [F5]
30C1:010B 8D161E01    LEA    DX,[011E]
30C1:010F CD21       INT    21
30C1:0111 8D165A01    LEA    DX,[015A]
30C1:0115 B409       MOV    AH,09
30C1:0117 CD21       INT    21
30C1:0119 B8004C     MOV    AX,4C00
30C1:011C CD21       INT    21
30C1:011E 8D167F01    LEA    DX,[017F]
-G119 [F5] ----- 終りまで実行

おじプログラムは実行されました。

さようなら。。。SYMDEB。。。----- 暴走してしまった

```

## ■システムコールを無効にする

### ○考え方

デバッガは、入力や出力を MS-DOS のシステムコールを用いて行っています。プログラムをトレースする際にもシステムコールを用いて、レジスタの内容や命令を表示しているのです。このシステムコールが機能しなければ、デバッガは何の情報も提供できなくなるのです。

### ○実現方法

基本的には INT3 の封じ込めと同じです。システムコールに対応する割り込みベクタは 21H ですから、ここを、自らが用意するルーチンへ書き換えてもよいでしょう。

### ○対処法

INT3 の封じ込めに対処するのと同様に、割り込みベクタを書き換えている箇所を潰すか、シングルステップ割り込みやインターバルタイマを用いて、割り込みベクタを書き改めます。

## ■シングルステップ割り込みを無効にする

### ○考え方

SYMDEB を用いてプログラムの実行を追跡する際に、前述のブレークポイントを置くものと、シングルステップ割り込みを用いた2通りの方法が考えられます。ここでは、後者の方法を封じる手段について紹介します。

### ○実現方法

SYMDEB は、随時、シングルステップ割り込みに対応するベクタを、正常なものへ書き戻すようなことを行っています。ですから、プログラム中でシングルステップに対抗した割り込みベクタを書き改めても、すぐに元に戻されて、トレースを停止させることができません。しかしこれを逆手にとって、書き戻すことをチェックするようにすれば、プログラムがSYMDEB 上で実行されていることを判断することができます。

実現のための方法はいたって簡単です。まずプログラムの先頭がある位置で、シングルステップ割り込みに対応したベクタ (INT 01H) を書き換えてみます。そしてしばらくした後に、そこが書き換えた内容と異なっていれば、SYMDEB 上で実行されていると判断することができるわけです。

## ○対処方法

SYMDEB を改造し、書き戻すという処理をキャンセルさせるしかありませんが、これでは、ほんとうにシングルステップ割り込みを封じる操作に出られてしまうこともあり、なかなか両方に対応する策というものはありません。しかし、書き戻す前に書き戻す操作が必要かということを調べれば、書き戻す操作が必要なときには、プログラム中で書き換えが行われたことがわかるわけですし、また必要でないときには、書き換えが行われていないことになります。よって、先手を打って相手の出方をうかがうという方法が考えられるのですが、SYMDEB 自体にパッチをあてるなど、多少実現には手間どります。

## ○サンプル

シングルステップ割り込みに対応した割り込みベクタを書き換え、きちんと書き換わったままになっているかどうかをチェックするプログラム TF.COM を、実行例と共に図 1.2 として示します。

■図 1.2 TF.ASM ソースリスト

```

:
: *****
:
:      TF.ASM
:
:      SYMDEBの機能を無効にするサンプル(2)
:
:      このプログラムを実行することによって、トレースフラグによる
:      ステップ実行の有無を検出することができます。
:
:      COPYRIGHT(C) 1987 BY SHUWA SYSTEM TRADING CO.,LTD.
:
:      LAST MODIFIED ON FEBRUARY 4TH,1987
:
: *****
:
CODE    SEGMENT
        ASSUME    CS:CODE,DS:CODE
:
:      ORG        100H
:
TF       PROC
:

```

---

```

        LEA     DX,MESSAGE      ; 警告を表示
        MOV     AH,9
        INT     21H
;
        MOV     AX,2501H
        LEA     DX,ENTRY
        INT     21H
;
        XOR     AX,AX           ; トレースされていないかチェック
        MOV     ES,AX
        MOV     BX,1           ; シングルステップ割り込みの割り込みタイプ
        SHL     BX,1
        SHL     BX,1
        MOV     AX,ES:[BX]      ; オフセットをチェック
        SUB     AX,OFFSET ENTRY
        MOV     DX,AX
        MOV     AX,ES:[BX+2]    ; セグメントをチェック
        MOV     CX,CS
        SUB     AX,CX
        OR      AX,DX
        JZ      NORMAL
;
        LEA     DX,BAD          ; トレースしていることを叱るメッセージ
        MOV     AH,9
        INT     21H
        JMP     EXIT
;
NORMAL:  LEA     DX,ENDING      ; おじに終了したことを告げるメッセージ
        MOV     AH,9
        INT     21H
;
EXIT:    MOV     AX,4C00H       ; プログラム終了
        INT     21H
;
ENTRY    PROC    FAR           ; シングルステップ割り込み処理ルーチン
;
        IRET                    ; ダミー
;
ENTRY    ENDP
;
MESSAGE  DB      13,10
        DB      'トレースはしていないでしょうね?'
        DB      13,10,'$'
;
ENDING   DB      13,10
        DB      'プログラムは正規に実行されました。'
        DB      13,10,'$'
;
BAD      DB      13,10
        DB      'あなたはトレースしていますね?'
        DB      13,10,'$'
;
TF       ENDP
;
CODE     ENDS
;
        END     TF

```

---



## ■図 1.2 TF.COM ダンプリスト

```

00000000 : 8D 16 49 01 B4 09 CD 21 B8 01 25 8D 16 48 01 CD : 52F
00000010 : 21 33 C0 8E C0 8B 01 00 D1 E3 D1 E3 26 88 07 2D : 768
00000020 : 48 01 88 D0 26 88 47 02 8C C9 2B C1 0B C2 74 0B : 628
00000030 : 8D 16 99 01 B4 09 CD 21 EB 09 90 8D 16 6E 01 B4 : 632
00000040 : 09 CD 21 B8 00 4C CD 21 CF 0D 0A 83 67 83 8C 81 : 649
00000050 : 5B 83 58 82 CD 82 B5 82 C4 82 A2 82 C8 82 A2 82 : 916
00000060 : C5 82 B5 82 E5 82 A4 82 CB 81 4B 0D 0A 24 0D 0A : 6F1
00000070 : 83 76 83 8D 83 4F 83 89 83 80 82 CD 90 83 8B 4B : 852
00000080 : 82 C9 8E C0 8D 73 82 B3 82 EA 82 C4 82 A2 82 DC : A02
00000090 : 82 B5 82 BD 81 44 0D 0A 24 0D 0A 82 A0 82 C8 82 : 67B
000000A0 : 8D 82 CD 83 67 83 8C 81 5B 83 58 82 B5 82 C4 82 : 8BB
000000B0 : A2 82 DC 82 B7 82 CB 81 4B 0D 0A 24 : 58A

```

## ■図 1.2 TF.COM 実行例

A>TF  ダイレクトに実行

トレースはしていないでしょうね？

プログラムは正規に実行されていました。———正常に実行される

A>SYMDEB B:¥CMDS¥TF.COM  デバッガで実行させてみる

```

Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086]

```

-U  念のため確認

```

30C1:0100 8D164901 LEA DX,[0149]
30C1:0104 B409 MOV AH,09
30C1:0106 CD21 INT 21
30C1:0108 B80125 MOV AX,2501
30C1:010B 8D164801 LEA DX,[0148]
30C1:010F CD21 INT 21
30C1:0111 33C0 XOR AX,AX
30C1:0113 8EC0 MOV ES,AX

```

-T100  トレースしてみる

```

AX=0000 BX=0000 CX=00BC DX=0149 SP=FFFE BP=0000 SI=0000 DI=0000
DS=30C1 ES=30C1 SS=30C1 CS=30C1 IP=0104 NV UP EI PL NZ NA PO NC
30C1:0104 B409 MOV AH,09
AX=0900 BX=0000 CX=00BC DX=0149 SP=FFFE BP=0000 SI=0000 DI=0000
DS=30C1 ES=30C1 SS=30C1 CS=30C1 IP=0106 NV UP EI PL NZ NA PO NC
30C1:0106 CD21 INT 21 ;Display String

```

トレースはしていないでしょうね？

```

AX=0924 BX=0000 CX=00BC DX=0149 SP=FFFE BP=0000 SI=0000 DI=0000
DS=30C1 ES=30C1 SS=30C1 CS=30C1 IP=0108 NV UP EI PL NZ NA PO NC
30C1:0108 B80125 MOV AX,2501
AX=2501 BX=0000 CX=00BC DX=0149 SP=FFFE BP=0000 SI=0000 DI=0000
DS=30C1 ES=30C1 SS=30C1 CS=30C1 IP=010B NV UP EI PL NZ NA PO NC
30C1:010B 8D164801 LEA DX,[0148] DS:0148=0DC
F
AX=2501 BX=0000 CX=00BC DX=0148 SP=FFFE BP=0000 SI=0000 DI=0000
DS=30C1 ES=30C1 SS=30C1 CS=30C1 IP=010F NV UP EI PL NZ NA PO NC
30C1:010F CD21 INT 21 ;Set Vector
AX=2501 BX=0000 CX=00BC DX=0148 SP=FFFE BP=0000 SI=0000 DI=0000
DS=30C1 ES=30C1 SS=30C1 CS=30C1 IP=0111 NV UP EI PL NZ NA PO NC
30C1:0111 33C0 XOR AX,AX
AX=0000 BX=0000 CX=00BC DX=0148 SP=FFFE BP=0000 SI=0000 DI=0000
DS=30C1 ES=30C1 SS=30C1 CS=30C1 IP=0113 NV UP EI PL NZ NA PE NC
30C1:0113 8EC0 MOV ES,AX
AX=0000 BX=0001 CX=00BC DX=0148 SP=FFFE BP=0000 SI=0000 DI=0000
DS=30C1 ES=0000 SS=30C1 CS=30C1 IP=0118 NV UP EI PL ZR NA PE NC
30C1:0118 D1E3 SHL BX,1

```

```

AX=0000 BX=0002 CX=00BC DX=0148 SP=FFFE BP=0000 SI=0000 DI=0000
DS=30C1 ES=0000 SS=30C1 CS=30C1 IP=011A NV UP EI PL NZ NA PO NC
30C1:011A D1E3 SHL BX,1
AX=0000 BX=0004 CX=00BC DX=0148 SP=FFFE BP=0000 SI=0000 DI=0000
DS=30C1 ES=0000 SS=30C1 CS=30C1 IP=011C NV UP EI PL NZ NA PO NC
30C1:011C 268B07 MOV AX,ES:[BX] ES:0004=0148
8
AX=2B45 BX=0004 CX=00BC DX=0148 SP=FFFE BP=0000 SI=0000 DI=0000
DS=30C1 ES=0000 SS=30C1 CS=30C1 IP=011F NV UP EI PL NZ NA PO NC
30C1:011F 2D4801 SUB AX,0148
AX=29FD BX=0004 CX=00BC DX=0148 SP=FFFE BP=0000 SI=0000 DI=0000
DS=30C1 ES=0000 SS=30C1 CS=30C1 IP=0122 NV UP EI PL NZ AC PO NC
30C1:0122 8BD0 MOV DX,AX
AX=29FD BX=0004 CX=00BC DX=29FD SP=FFFE BP=0000 SI=0000 DI=0000
DS=30C1 ES=0000 SS=30C1 CS=30C1 IP=0124 NV UP EI PL NZ AC PO NC
30C1:0124 268B4702 MOV AX,ES:[BX+02] ES:0006=27C8
8
AX=27D8 BX=0004 CX=00BC DX=29FD SP=FFFE BP=0000 SI=0000 DI=0000
DS=30C1 ES=0000 SS=30C1 CS=30C1 IP=0128 NV UP EI PL NZ AC PO NC
30C1:0128 8CC9 MOV CX,CS
AX=F717 BX=0004 CX=30C1 DX=29FD SP=FFFE BP=0000 SI=0000 DI=0000
DS=30C1 ES=0000 SS=30C1 CS=30C1 IP=012C NV UP EI NG NZ NA PE CY
30C1:012C 0BC2 OR AX,DX
AX=FFFF BX=0004 CX=30C1 DX=29FD SP=FFFE BP=0000 SI=0000 DI=0000
DS=30C1 ES=0000 SS=30C1 CS=30C1 IP=012E NV UP EI NG NZ NA PE NC
30C1:012E 7408 JZ 0138
AX=FFFF BX=0004 CX=30C1 DX=29FD SP=FFFE BP=0000 SI=0000 DI=0000
DS=30C1 ES=0000 SS=30C1 CS=30C1 IP=0130 NV UP EI NG NZ NA PE NC
30C1:0130 8D169901 LEA DX,[0199] DS:0199=0A0D
D
AX=FFFF BX=0004 CX=30C1 DX=0199 SP=FFFE BP=0000 SI=0000 DI=0000
DS=30C1 ES=0000 SS=30C1 CS=30C1 IP=0134 NV UP EI NG NZ NA PE NC
30C1:0134 B409 MOV AH,09
AX=09FF BX=0004 CX=30C1 DX=0199 SP=FFFE BP=0000 SI=0000 DI=0000
DS=30C1 ES=0000 SS=30C1 CS=30C1 IP=0136 NV UP EI NG NZ NA PE NC
30C1:0136 CD21 INT 21 ;Display String

```

あなたはトレースしていますね? —— -トレースしているのがわかってしまう

```

AX=0924 BX=0004 CX=30C1 DX=0199 SP=FFFE BP=0000 SI=0000 DI=0000
DS=30C1 ES=0000 SS=30C1 CS=30C1 IP=0138 NV UP EI NG NZ NA PE NC
30C1:0138 E809 JMP 0143
AX=0924 BX=0004 CX=30C1 DX=0199 SP=FFFE BP=0000 SI=0000 DI=0000
DS=30C1 ES=0000 SS=30C1 CS=30C1 IP=0143 NV UP EI NG NZ NA PE NC
30C1:0143 88004C MOV AX,4C00
AX=4C00 BX=0004 CX=30C1 DX=0199 SP=FFFE BP=0000 SI=0000 DI=0000
DS=30C1 ES=0000 SS=30C1 CS=30C1 IP=0146 NV UP EI NG NZ NA PE NC
30C1:0146 CD21 INT 21 ;Terminate a Process

```

Program terminated normally (0)

-Q[9]

A>

## 1.2 親をチェックする

### ○考え方

MS-DOS では、プログラムはプロセスとして起動しています。このときプロセスを起動したほうを親プロセス、起動されたほうを子プロセスといいます。一般的には、アプリケーションは COMMAND.COM が子プロセスとして起動するのですから、親が COMMAND.COM であるかどうかをチェックすると、デバッガで起動した場合には、チェック失敗ということになって、アプリケーション側では、何かしらの手が打てることになります。

### ○実現方法

実際に親を調べるには、親の環境をチェックする方法があります。環境というのは、各プロセスに個別に用意されている文字列の集合体ですが、その内容は SET コマンドで見ることができます。SET コマンドを実行すると、だいたい、次のように環境の内容が表示されるはずです。

```
A>set
COMSPEC =¥COMMAND.COM
PATH =
```

この場合、表示されるのは COMMAND.COM の環境ですが、この表示は COMMAND.COM が自らの環境を表示しているからにはかなりません。同様にあらゆるプロセスが、自らの環境を表示しないまでも、参照することができるのです。

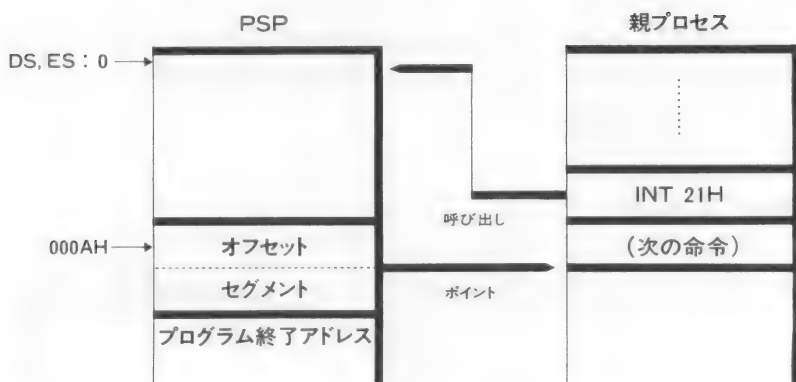
環境の位置というのは PSP に格納されています。プロセスが起動したときにはレジスタ DS, ES は、PSP のセグメント（オフセットは 0000H）を指していますから、環境セグメント（オフセットは 0000H）が格納してあるオフセット 002CH を参照すればよいわけです（図 1.3 参照）。

■図 1.3 環境



しかし、自らが参照できる環境は自らの PSP に入っているものですから、環境をチェックしたところで親を知る術はありません。そこで、PSP に格納されている別の情報、プログラム終了アドレス（子プロセスが終了したときに戻るべき、親プロセスでの位置が記録されている）を参照して、親の PSP の位置を求めます。具体的には、PSP のオフセット 0000AH を参照すればよく、そこにプロセスが終了したときに戻るべきアドレスが、セグメント：オフセットと対で格納されていますから、そのセグメント部分を参照すれば親のセグメントがわかるわけです。親が COMMAND.COM ならば、戻りアドレスのセグメントを、そのまま PSP のセグメントとみなすことができます（図 1.4 参照）。

■図 1.4 プログラム終了アドレス



さて親の PSP の位置がわかったら、自らの環境を参照するのと同じ手順で環境を参照してみましょう（オフセット 002CH を見てみます）。ここが 0000H であれば親は最初に起動された COM-MAND.COM です。SYMDEB などのデバッガから起動すると親のプロセスの環境は、デバッガが故意にオフセット 002CH を 0000H にしていない限り、デバッガから起動されていることを見抜かれてしまうわけです。

ここで、親の環境をチェックする簡単なサブルーチン GET - PARENT - ENV を紹介します。これは、自らの PSP のセグメントがレジスタ DS に入っているものとして、親の環境セグメントをレジスタ AX に持ってくるものです。

#### GET - PARENT - ENV PROC NEAR

PUSH ES

MOV ES,DS: [0AH+2]; プログラム中断アドレスを取得

MOV AX,ES: [2CH]; 親の環境セグメントを取得

```

POP     ES
RET
GET-PARENT-ENV ENDP

```

非常に簡単なプログラムですが、このプログラムをサブルーチンとして CALL したあと、レジスタ AX の内容をチェックすれば、親が COMMAND.COM（それも大元の）であるかどうかを容易にチェックすることができます。

### ○対処方法

SYMDEB には環境は特に必要ないので、自らの PSP において、環境アドレスを強引に 0000H に設定してしまいましょう。

方法は、GET-PARENT-ENV と同様の原理で逆のを行います。すなわち、SYMDEB 起動直後にプロンプトが表示されている状態で、

```
-D 000A [F2]
```

（子の PSP におけるプログラム終了アドレスがダンプされる）

```
-E XXXX:2C 00 00 [F2] ← XXXX はプログラム終了アドレスの  
セグメント
```

とすれば OK です。

### ○サンプル

親の環境をチェックし、COMMAND.COM でなければ、その旨のメッセージを表示して終了するプログラム、CHKPAR.COM を図 1.5 として示します。

## ■図 1.5 CHKPAR.ASM ソースリスト

```

;
; *****
;
;      CHKPAR.ASM
;
;      SYMDEBの機能を無効にするサンプル(3)
;
;      このプログラムは、親がおもとのCOMMAND.COMであるかどうか
;      チェックします。
;
;      COPYRIGHT(C) 1987 BY SHUWA SYSTEM TRADING CO.,LTD.
;
;      LAST MODIFIED ON FEBRUARY 4TH,1987
;
; *****
CODE    SEGMENT
        ASSUME    CS:CODE,DS:CODE
;
;      ORG        100H
;
CHKPAR  PROC
;
;      LEA        DX,MESSAGE      ; 警告を表示
;      MOV        AH,9
;      INT        21H
;
;      MOV        ES,DS:[0CH]
;      MOV        AX,ES:[2CH]
;      OR         AX,AX
;      JZ         NORMAL
;
;      LEA        DX,BAD          ; デバッガから起動したことを叱るメッセージ
;      MOV        AH,9
;      INT        21H
;      JMP        EXIT
;
NORMAL:
;      LEA        DX,ENDING       ; おじに終了したことを告げるメッセージ
;      MOV        AH,9
;      INT        21H
;
EXIT:
;      MOV        AX,4C00H        ; プログラム終了
;      INT        21H
;
MESSAGE DB      13,10
        DB      'デバッガから動かしていませんよね?'
        DB      13,10,'$'
;
ENDING  DB      13,10
        DB      'プログラムは正規に実行されました。'
        DB      13,10,'$'
;
BAD      DB      13,10
        DB      'あなたはデバッガから動かしていますね?'
        DB      13,10,'$'
;
CHKPAR  ENDP
;
CODE    ENDS
;
        END        CHKPAR

```

## ■図 1.5 CHKPAR.COM ダンプリスト

```

00000000 : 8D 16 2C 01 B4 09 CD 21 8E 06 0C 00 26 A1 2C 00 : 40E
00000010 : 08 C0 74 08 8D 16 7E 01 B4 09 CD 21 EB 09 90 8D : 628
00000020 : 16 57 01 B4 09 CD 21 B8 00 4C CD 21 0D 0A 83 66 : 508
00000030 : 83 6F 83 62 83 48 82 A9 82 E7 93 AE 82 A9 82 B5 : 8DC
00000040 : 82 C4 82 A2 82 C8 82 A2 82 C5 82 B5 82 E5 82 A4 : 9E3
00000050 : 82 C8 81 48 0D 0A 24 0D 0A 83 76 83 8D 83 4F 83 : 5C6
00000060 : 89 83 80 82 CD 90 B3 88 48 82 C9 8E C0 8D 73 82 : 90F
00000070 : B3 82 EA 82 DC 82 B5 82 BD 81 44 0D 0A 24 0D 0A : 70A
00000080 : 82 A0 82 C8 82 BD 82 CD 83 66 83 6F 83 62 83 4B : 888
00000090 : 82 A9 82 E7 93 AE 82 A9 82 B5 82 C4 82 A2 82 DC : 9FF
000000A0 : 82 B7 82 CB 81 48 0D 0A 24 : 38A

```

## ■図 1.5 CHKPAR.COM 実行例

A>CHKPAR ————ダイレクトに実行させる

デバッグから動かしていないでしょうね？

プログラムは正規に実行されました。——正常に実行される

A>SYMDEB B:¥CMDS¥CHKPAR.COM ————デバッグで動作させてみる

Microsoft Symbolic Debug Utility

Version 3.01

(C)Copyright Microsoft Corp 1984, 1985

Processor is [8086]

-U ————確認

```

30C1:0100 8D162C01      LEA    DX,[012C]
30C1:0104 B409          MOV    AH,09
30C1:0106 CD21          INT     21
30C1:0108 8E060C00      MOV    ES,[000C]
30C1:010C 26A12C00      MOV    AX,ES:[002C]
30C1:0110 0BC0          OR     AX,AX
30C1:0112 740B          JZ     011F
30C1:0114 8D167E01      LEA    DX,[017E]

```

-U ————確認

```

30C1:0118 B409          MOV    AH,09
30C1:011A CD21          INT     21
30C1:011C EB09          JMP     0127
30C1:011E 90             NOP
30C1:011F 8D165701      LEA    DX,[0157]
30C1:0123 B409          MOV    AH,09
30C1:0125 CD21          INT     21
30C1:0127 B8004C      MOV    AX,4C00

```

-G=100,127 ————一気に実行

デバッグから動かしていないでしょうね？

あなたはデバッグから動かしていますね？——ばれてしまう

```

AX=0924 BX=0000 CX=00A9 DX=017E SP=FFFE BP=0000 SI=0000 DI=0000
DS=30C1 ES=27D8 SS=30C1 CS=30C1 IP=0127 NV UP EI PL NZ NA PE NC
30C1:0127 B8004C      MOV    AX,4C00

```

-Q ————確認

A>



## 1.3 実行時間をチェックする

### ■カレンダー時計を用いる

#### ○考え方

デバッガでプログラムを起動し、それをトレースしたりブレークポイントを置いて実行を追っていると、実際に実行される時間より長めに時間が費やされます。これはトレースの際にレジスタの内容や命令を表示したり、またブレークポイントが置かれることによって実行が小刻みに行われ、実行時間に隙間ができるからです。

これを利用すれば、デバッガによる実行追跡をチェックすることができます。

#### ○実現方法

プログラム中の2箇所のポイントにおいて、まず最初のポイントで日付と時刻を記録し、終りのポイントで日付と時刻から先ほど記録しておいた日付と時刻を引き算します。得られた結果が規定範囲外にあれば、なんらかの実行中断があったと判断できるわけです。

具体的には日付と時刻を取得するサブルーチン、それらを秒などの単位に換算するサブルーチン、引き算するサブルーチンがあれば実現できます。

#### ○対処方法

チェックを行っている箇所を捜し出して潰すしかないでしょう。日時の読み取りを BIOS を用いて行っているとすれば、そこをロギングする方法もあります。

## ○サンプル

プログラムの実行開始時刻と実行終了時刻を比較し、規定値内に入っていないければ無限ループに陥るプログラム CHKEXE1.COM を、実行例とともに図 1.6 として示します。トレースを行ったリブレークポイントを置いて実行した場合には、必ず無限ループに陥りますので注意してください。

■図 1.6 CHKEXE1. ASM ソースリスト

```

:
:*****
:
:      CHKEXE1.ASM
:
:      SYMDEBの機能を無効にするサンプル(4)
:
:      このプログラムは、実行時間をカレンダー時計によって計測し、
:      実行が正規に行われているかチェックします。
:
:      注意！
:      ○実行が正規に行われていない場合、無限ループに陥ります。
:      ○CPUは80286を使用しないで下さい。
:      ○動作中にクロック周波数を切り替えるのはやめてください。
:      ○インターバルタイマなどを多用している場合、正常に動作しない
:      場合が考えられます(PRINTなど)。
:
:      COPYRIGHT(C) 1987 BY SHUWA SYSTEM TRADING CO.,LTD.
:
:      LAST MODIFIED ON FEBRUARY 4TH,1987
:
:*****
:
CODE    SEGMENT
        ASSUME    CS:CODE,DS:CODE
:
:      ORG        100H
:
CHKEXE1 PROC
CALL     GET_RANGE      ; 許容範囲を得る
MOV      AH,2CH         ; 開始時の時刻を読む
INT      21H
MOV      BEGIN_MIN,CL   ; 時刻をセーブ
MOV      BEGIN_SEC,DH
:
:      LEA        DX,MESSAGE      ; 警告を表示
:      MOV        AH,9
:      INT        21H
:
:      MOV        CX,200          ; ダミーループ
:
DUMMY_LOOP:
        PUSH     CX
        XOR      CX,CX
        LOOP     $
        POP      CX
        LOOP     DUMMY_LOOP

```

```

;
MOV     AH,2CH           ; 終了時の時刻を読み取る
INT     21H
PUSH    DX               ; 秒を待避
MOV     AL,CL            ; 分を秒に換算
XOR     AH,AH
MOV     DX,60
MUL     DX
POP     DX
XCHG    DL,DH
ADD     AX,DX            ; 分と秒を加える
PUSH    AX

;
MOV     AL,BEGIN_MIN     ; 分を秒に換算
XOR     AH,AH
MOV     DX,60
MUL     DX
MOV     DL,BEGIN_SEC
XOR     DH,DH
ADD     DX,AX            ; 秒を加える
POP     AX

;
SUB     AX,DX            ; 所要時間を算出
CMP     AX,RANGE_TOP     ; 規定値内に入るか見る
JA      ABNORMAL         ; 大きすぎたらだめ

;
CMP     AX,RANGE_BOTTOM  ;
JNB     NORMAL           ; 小さすぎてもだめ

ABNORMAL:
LEA     DX,BAD           ; プログラムを小刻みに実行したことを
MOV     AH,9             ; 叱るメッセージ
INT     21H
JMP     $                ; 無限ループ

;
NORMAL:
LEA     DX,ENDING        ; ぶじに終了したことを告げるメッセージ
MOV     AH,9
INT     21H

;
EXIT:
MOV     AX,4C00H         ; プログラム終了
INT     21H

;
GET_RANGE PROC NEAR     ; 許容範囲を算出
PUSH    ES
XOR     AX,AX
MOV     ES,AX
MOV     AL,ES:[501H]      ; システム情報を取り出す
AND     AL,11000000B      ; クロック周波数・CPUを判別する
CMP     AL,01000000B      ; V30,10MHzか?
JE      GET_RANGE_EXIT

;
CMP     AL,11000000B      ; V30,8MHzか?
JNE     GET_RANGE_1

;
MOV     RANGE_TOP,39      ; V30,8MHz用の定数をセット
MOV     RANGE_BOTTOM,36
GET_RANGE_EXIT

;
GET_RANGE_1:
CMP     AL,10000000B      ; 8086,8MHzか?
JNE     GET_RANGE_2

;
MOV     RANGE_TOP,39      ; 8086,8MHz用の定数をセット
MOV     RANGE_BOTTOM,35
GET_RANGE_EXIT

```

```

;
GET_RANGE_2:
    MOV     RANGE_TOP,53      ; 8086,5MHz用の定数をセット
    MOV     RANGE_BOTTOM,49
;
GET_RANGE_EXIT:
    POP     ES
    RET
GET_RANGE    ENDP
;
RANGE_TOP    DW      32      ; 実行に要する最大秒 (初期値V30/10MHz)
RANGE_BOTTOM DW      28      ; 実行に要する最小秒 (初期値V30/10MHz)
;
BEGIN_MIN    DB      ?       ; チェック開始分を格納
BEGIN_SEC    DB      ?       ; チェック開始秒を格納
;
MESSAGE DB
        13,10
        'プログラムは一気に実行させて下さいね。'
        DB
        13,10,'$'
;
ENDING DB
        13,10
        'プログラムは正規に実行されました。'
        DB
        13,10,'$'
;
BAD DB
        13,10
        'あなたはプログラムを止めたね?'
        DB
        13,10,'$'
;
CHKEXE1 ENDP
;
CODE ENDS
;
END      CHKEXE1

```

■ 図 1.6 CHKEXE1.COM ダンプリスト

```

00000000 : E8 68 00 B4 2C CD 21 88 0E B5 01 88 36 B6 01 8D : 66F
00000010 : 16 B7 01 B4 09 CD 21 B9 C8 00 51 33 C9 E2 FE 59 : 780
00000020 : E2 F8 B4 2C CD 21 52 8A C1 32 E4 BA 3C 00 F7 E2 : 92A
00000030 : 5A 86 D6 03 C2 50 A0 B5 01 32 E4 BA 3C 00 F7 E2 : 806
00000040 : 8A 16 B6 01 32 F6 03 D0 58 2B C2 3B 06 B1 01 77 : 601
00000050 : 06 3B 06 B3 01 73 0A 8D 16 09 02 B4 09 CD 21 E8 : 4BC
00000060 : FE 8D 16 E2 01 B4 09 CD 21 88 00 4C CD 21 06 33 : 65A
00000070 : C0 8E C0 26 A0 01 05 24 C0 3C 40 74 32 3C C0 75 : 651
00000080 : 0F C7 06 B1 01 27 00 C7 06 B3 01 24 00 EB 20 90 : 4F5
00000090 : 3C 80 75 0F C7 06 B1 01 27 00 C7 06 B3 01 23 00 : 48A
000000A0 : EB 0D 90 C7 06 B1 01 35 00 C7 06 B3 01 31 00 07 : 4F5
000000B0 : C3 20 00 1C 00 00 00 0D 0A 83 76 83 8D 83 4F 83 : 474
000000C0 : 89 83 80 82 CD 88 EA 8B 43 82 C9 8E C0 8D 73 82 : 936
000000D0 : B3 82 B9 82 C4 89 BA 82 B3 82 A2 82 CB 81 44 0D : 8EF
000000E0 : 0A 24 0D 0A 83 76 83 8D 83 4F 83 89 83 80 82 CD : 67E
000000F0 : 90 B3 88 48 82 C9 8E C0 8D 73 82 B3 82 EA 82 DC : 9B1
00000100 : 82 B5 82 BD 81 44 0D 0A 24 0D 0A 82 A0 82 C8 82 : 67B
00000110 : BD 82 CD 83 76 83 8D 83 4F 83 89 83 80 82 F0 8E : 8F6
00000120 : 7E 82 DF 82 DC 82 B5 82 BD 82 CB 81 48 0D 0A 24 : 804

```

## ■ 図 1.6 CHKEXE1.COM 実行例

A>CHKEXE1[

——ダイレクトに実行

プログラムは一気に実行させて下さいね。

プログラムは正規に実行されました。——規定時間内に実行できる

A>SYMDEB B:¥CMD\$¥CHKEXE1.COM[

——デバッガで動作させてみる

Microsoft Symbolic Debug Utility

Version 3.01

(C)Copyright Microsoft Corp 1984, 1985

Processor is [8086]

-U[

```

30C5:0100 E86B00      CALL    016E
30C5:0103 B42C        MOV     AH,2C
30C5:0105 CD21        INT     21
30C5:0107 880E8501    MOV     [01B5],CL
30C5:0108 8B36B601    MOV     [01B6],DH
30C5:010F 8D16B701    LEA     DX,[01B7]
30C5:0113 B409        MOV     AH,09
30C5:0115 CD21        INT     21
-U[]
30C5:0117 B9C800      MOV     CX,00C8
30C5:011A 51          PUSH    CX
30C5:011B 33C9        XOR     CX,CX
30C5:011D E2FE        LOOP   011D
30C5:011F 59          POP     CX
30C5:0120 E2F8        LOOP   011A
30C5:0122 B42C        MOV     AH,2C
30C5:0124 CD21        INT     21

```

-T100[

——トレースしながら実行

```

AX=0000 BX=0000 CX=0130 DX=0000 SP=FFFC BP=0000 SI=0000 DI=0000
DS=30C5 ES=30C5 SS=30C5 CS=30C5 IP=016E NV UP EI PL NZ NA PO NC
30C5:016E 06          PUSH    ES
AX=0000 BX=0000 CX=0130 DX=0000 SP=FFFA BP=0000 SI=0000 DI=0000
DS=30C5 ES=30C5 SS=30C5 CS=30C5 IP=016F NV UP EI PL NZ NA PO NC
30C5:016F 33C0        XOR     AX,AX
AX=0000 BX=0000 CX=0130 DX=0000 SP=FFFA BP=0000 SI=0000 DI=0000
DS=30C5 ES=30C5 SS=30C5 CS=30C5 IP=0171 NV UP EI PL ZR NA PE NC
30C5:0171 8EC0        MOV     ES,AX
AX=0064 BX=0000 CX=0130 DX=0000 SP=FFFA BP=0000 SI=0000 DI=0000
DS=30C5 ES=0000 SS=30C5 CS=30C5 IP=0177 NV UP EI PL ZR NA PE NC
30C5:0177 24C0        AND     AL,C0
AX=0040 BX=0000 CX=0130 DX=0000 SP=FFFA BP=0000 SI=0000 DI=0000
DS=30C5 ES=0000 SS=30C5 CS=30C5 IP=0179 NV UP EI PL NZ NA PO NC
30C5:0179 3C40        CMP     AL,40
AX=0040 BX=0000 CX=0130 DX=0000 SP=FFFA BP=0000 SI=0000 DI=0000
DS=30C5 ES=0000 SS=30C5 CS=30C5 IP=017B NV UP EI PL ZR NA PE NC
30C5:017B 7432        JZ      01AF
AX=0040 BX=0000 CX=0130 DX=0000 SP=FFFA BP=0000 SI=0000 DI=0000
DS=30C5 ES=0000 SS=30C5 CS=30C5 IP=01AF NV UP EI PL ZR NA PE NC
30C5:01AF 07          POP     ES
AX=0040 BX=0000 CX=0130 DX=0000 SP=FFFE BP=0000 SI=0000 DI=0000
DS=30C5 ES=30C5 SS=30C5 CS=30C5 IP=0103 NV UP EI PL ZR NA PE NC
30C5:0103 B42C        MOV     AH,2C
AX=2C40 BX=0000 CX=0130 DX=0000 SP=FFFE BP=0000 SI=0000 DI=0000
DS=30C5 ES=30C5 SS=30C5 CS=30C5 IP=0105 NV UP EI PL ZR NA PE NC
30C5:0105 CD21        INT     21 ;Get Time
AX=2C00 BX=0000 CX=0D05 DX=0500 SP=FFFE BP=0000 SI=0000 DI=0000
DS=30C5 ES=30C5 SS=30C5 CS=30C5 IP=0107 NV UP EI PL ZR NA PE NC
30C5:0107 880E8501    MOV     [01B5],CL DS:01B5=00
AX=2C00 BX=0000 CX=0D05 DX=0500 SP=FFFE BP=0000 SI=0000 DI=0000
DS=30C5 ES=30C5 SS=30C5 CS=30C5 IP=010B NV UP EI PL ZR NA PE NC
30C5:010B 8B36B601    MOV     [01B6],DH DS:01B6=00
AX=2C00 BX=0000 CX=0D05 DX=0500 SP=FFFE BP=0000 SI=0000 DI=0000
DS=30C5 ES=30C5 SS=30C5 CS=30C5 IP=010F NV UP EI PL ZR NA PE NC
30C5:010F 8D16B701    LEA     DX,[01B7] DS:01B7=0A0

```

156 ■ 応用編II

```

D
AX=2C00 BX=0000 CX=0D05 DX=01B7 SP=FFFE BP=0000 SI=0000 DI=0000
DS=30C5 ES=30C5 SS=30C5 CS=30C5 IP=0113 NV UP EI PL ZR NA PE NC
30C5:0113 B409 MOV AH,09
AX=0900 BX=0000 CX=0D05 DX=01B7 SP=FFFE BP=0000 SI=0000 DI=0000
DS=30C5 ES=30C5 SS=30C5 CS=30C5 IP=0115 NV UP EI PL ZR NA PE NC
30C5:0115 CD21 INT 21 ;Display String

```

プログラムは一気に実行させて下さいね。

```

AX=0924 BX=0000 CX=0D05 DX=01B7 SP=FFFE BP=0000 SI=0000 DI=0000
DS=30C5 ES=30C5 SS=30C5 CS=30C5 IP=0117 NV UP EI PL ZR NA PE NC
30C5:0117 B9C800 MOV CX,00C8
AX=0924 BX=0000 CX=00C8 DX=01B7 SP=FFFE BP=0000 SI=0000 DI=0000
DS=30C5 ES=30C5 SS=30C5 CS=30C5 IP=011A NV UP EI PL ZR NA PE NC
30C5:011A 51 PUSH CX
AX=0924 BX=0000 CX=00C8 DX=01B7 SP=FFFC BP=0000 SI=0000 DI=0000
DS=30C5 ES=30C5 SS=30C5 CS=30C5 IP=011B NV UP EI PL ZR NA PE NC
30C5:011B 33C9 XOR CX,CX
AX=0924 BX=0000 CX=0000 DX=01B7 SP=FFFC BP=0000 SI=0000 DI=0000
DS=30C5 ES=30C5 SS=30C5 CS=30C5 IP=011D NV UP EI PL ZR NA PE NC
30C5:011D E2FE LOOP 011D

```

ルーフを切る

```

-U
30C5:011F 59 POP CX
30C5:0120 E2F8 LOOP 011A
30C5:0122 B42C MOV AH,2C
30C5:0124 CD21 INT 21
30C5:0126 52 PUSH DX
30C5:0127 8AC1 MOV AL,CL
30C5:0129 32E4 XOR AH,AH
30C5:012B BA3C00 MOV DX,003C

```

終わりまで一気に実行

```

-G122
AX=0924 BX=0000 CX=0000 DX=01B7 SP=FFFE BP=0000 SI=0000 DI=0000
DS=30C5 ES=30C5 SS=30C5 CS=30C5 IP=0122 NV UP EI PL ZR NA PE NC
30C5:0122 B42C MOV AH,2C
-G

```

あなたはプログラムを止めましたね?—時間がかかりすぎた

## ■インターバルタイマを用いる

### ○考え方

日付と時刻を読み取って時間差を求めるという静的な方法に対して、こちらは動的に時間を計測するものです。カレンダー時計を用いる場合と同様に、実際に、実行される時間が長いため、指定時間内にはプログラムが実行できないことを利用します。

### ○実現方法

周期的に入る割り込み（インターバルタイマ割り込み）を検出し、プログラム中の2箇所以上のポイントで、実行中に何回割り込みが

入ったかをカウントします。回数が規定範囲外にあれば実行中断とみなせます。

具体的には、プログラムの先頭でタイマを初期化しておき、割り込みの入るたびにカウンタを増加します。次に、チェックポイントで割り込みを停止させカウンタの内容を参照します。以上の手順を書き表せば、次のようになるでしょう。

```

.....

UPPER    EQU    100
LOWER    EQU    200

.....

TIME_COUNT    DW    0    ; タイムカウンタ

.....

MOV        TIME_COUNT,0 ; カウンタのクリア
CALL       INIT_TIMER   ; タイマの初期化

.....                                ; チェックアウト前に行うべき処理

CALL       ABORT_TIMER ; タイマの停止
CMP        COUNT,LOWER ; 少ないか？
JB         ABORT
CMP        COUNT,UPPER ; 多いか？
JA         ABORT

.....                                ; 引続き正常な処理を行う

ABORT:     JMP     BORT    ; 無限ループに陥る

.....

```

プログラムの流れを特に説明する必要はないでしょうが、いちおう説明しておきますと、まず設けられている変数 TIME- COUNT は、インターバルタイマが入るたびに増加し、時間経過をダイナミックに計測します。処理が終ればタイマは停止され TIME- COUNT が参照されます。このとき、TIME- COUNT が UPPER と LOWER の範囲になれば、無限ループに陥ります。

### ○対処方法

カレンダー時計を用いている場合と同様、チェックを行っている箇所を捜し出し潰すしかないでしょう。インターバルタイマの動作を停止させてしまうという方法もあります。

### ○サンプル

インターバルタイマによって時間を計測し、規定時間に入っているかどうかを調べ、入っていなければ無限ループに陥るプログラム CHKEXE2.COM を、実行例と共に図 1.7 として示します。

トレースを行ったりブレークポイントを置いて実行した場合には、間違いなく無限ループに陥りますので注意してください。

■図 1.7 CHKEXE2. ASM ソースリスト

```

:
: *****
:
:      CHKEXE2. ASM
:
:      SYMDEB の機能を無効にするサンプル ( 5 )
:
:      このプログラムは、実行時間をインターバルタイマによって計測し、
:      実行が正規に行われているかチェックします。
:      注意！
:      ○実行が正規に行われていない場合、無限ループに陥ります。
:      ○CPUは80286を使用しないで下さい。
:      ○動作中にクロック周波数を切り替えるようなことはしないで
:      下さい。
:      ○インターバルタイマなどを多用した場合には、正常に動作しない
:      ことがあります。
:
:      COPYRIGHT(C) 1987 BY SHUWA SYSTEM TRADING CO.,LTD.

```



```

;
;      LAST MODIFIED ON FEBRUARY 5TH,1987
;
;*****
;
VECTORS SEGMENT AT 0
;
;      ORG      7*4
;
VECTOR7 DD      ?      ; タイムアウト時に呼ばれるエントリ
VECTOR8 DD      ?      ; ハードウェアによって発生するタイマ割り込み
;
VECTORS ENDS
;
CODE      SEGMENT
ASSUME    CS:CODE,DS:CODE,ES:CODE,SS:CODE
;
;      ORG      100H
;
CHKEXE2 PROC
;
;      JMP      MAIN
;
VECTOR_SAVE7 DD      ?      ; 本来の INT 07Hベクタ
VECTOR_SAVE8 DD      ?      ; 本来の INT 08Hベクタ
TIME_COUNT   DW      ?      ; 秒カウンタ
;
RANGE_TOP    DW      32*100 ; 実行に要する最大秒 (10ms,初期値V30/10MHz)
RANGE_BOTTOM DW      28*100 ; 実行に要する最小秒 (10ms,初期値V30/10MHz)
;
MAIN:
CALL        GET_RANGE      ; 許容範囲を得る
MOV         TIME_COUNT,0   ; 秒カウンタを初期化する
CALL        INIT_TIMER     ; インターバルタイマを初期化する
;
LEA         DX,MESSAGE     ; 警告を表示
MOV         AH,9
INT         21H
;
MOV         CX,200         ; ダミーループ
;
DUMMY_LOOP:
PUSH        CX
XOR         CX,CX
LOOP        $
POP         CX
LOOP        DUMMY_LOOP
;
CALL        ABORT_TIMER    ; インターバルタイマの停止
MOV         AX,TIME_COUNT  ; 所要時間を取り出す
CMP         AX,RANGE_TOP   ; 規定値内に入るか見る
JA          ABNORMAL      ; 大きすぎたらだめ
;
CMP         AX,RANGE_BOTTOM
JNB         NORMAL        ; 小さすぎてもだめ
;
ABNORMAL:
LEA         DX,BAD         ; プログラムを小刻みに実行したことを
MOV         AH,9           ; 叱るメッセージ
INT         21H
JMP         $              ; 無限ループ
;
NORMAL:
LEA         DX,ENDING      ; おじに終了したことを告げるメッセージ
MOV         AH,9
INT         21H

```

```

;
EXIT:  MOV     AX,4C00H      ; プログラム終了
      INT     21H

;
INIT_TIMER PROC NEAR      ; インターバルタイマの初期化
      ASSUME DS:VECTORS,ES:NOTHING
      STI
      PUSH    DS
      PUSH    ES
      XOR     AX,AX
      MOV     DS,AX
      LES     AX,VECTOR8    ; 本来の INT 08H に対する割り込みベクタを取り出
;
      MOV     WORD PTR VECTOR_SAVE8,AX      ; オフセットをセーブ
      MOV     WORD PTR VECTOR_SAVE8+2,ES    ; セグメントをセーブ
;
      LES     AX,VECTOR7    ; 本来の INT 07H に対する割り込みベクタを取り出
;
      MOV     WORD PTR VECTOR_SAVE7,AX      ; オフセットをセーブ
      MOV     WORD PTR VECTOR_SAVE7+2,ES    ; セグメントをセーブ
;
      MOV     WORD PTR VECTOR8,OFFSET COUNT_UP      ; ベクタを設定する
      MOV     WORD PTR VECTOR8+2,CS
;
      POP     ES
      MOV     AH,2          ; インターバルタイマの起動
      MOV     CX,0          ; ダミー (時間を最大に設定)
      LEA     BX,NULL      ; ダミー
      INT     1CH
      POP     DS
      RET
INIT_TIMER ENDP

;
ABORT_TIMER PROC NEAR      ; インターバルタイマの停止
      ASSUME DS:VECTORS,ES:NOTHING
      STI
      PUSH    DS
      PUSH    ES
      IN      AL,02H        ; タイマ割り込みを禁止
      OR      AL,1
      OUT     02H,AL
      XOR     AX,AX
      MOV     DS,AX
      LES     AX,VECTOR_SAVE7 ; 本来のベクタに値を戻す
      MOV     WORD PTR VECTOR7,AX
      MOV     WORD PTR VECTOR7+2,ES
      LES     AX,VECTOR_SAVE8 ; 本来のベクタに値を戻す
      MOV     WORD PTR VECTOR8,AX
      MOV     WORD PTR VECTOR8+2,ES
      POP     ES
      POP     DS
      RET
ABORT_TIMER ENDP

;
COUNT_UP PROC              ; 1秒ごとに呼ばれるルーチン
      ASSUME DS:NOTHING,ES:NOTHING,SS:NOTHING
      STI
      PUSH    AX
      INC     TIME_COUNT    ; 秒カウンタをアップさせる
      MOV     AL,20H
      OUT     00H,AL        ; EOI 処理
      POP     AX
      IRET
COUNT_UP ENDP

;
NULL PROC                  ; タイムアウト時に呼ばれる (INT 07H) が無視

```

```

; グミー処理
NULL      IRET
ENDP

;
GET_RANGE PROC    NEAR    ; 許容範囲を算出
    PUSH    ES
    XOR     AX,AX
    MOV     ES,AX
    MOV     AL,ES:[501H]    ; システム情報を取り出す
    AND     AL,11000000B    ; クロック周波数・CPUを判別する
    CMP     AL,01000000B    ; V30,10MHzか?
    JE      GET_RANGE_EXIT

;
    CMP     AL,11000000B    ; V30,8MHzか?
    JNE     GET_RANGE_1

;
    MOV     RANGE_TOP,39*100    ; V30,8MHz用の定数をセット
    MOV     RANGE_BOTTOM,36*100
    JMP     GET_RANGE_EXIT

;
GET_RANGE_1:
    CMP     AL,10000000B    ; 8086,8MHzか?
    JNE     GET_RANGE_2

;
    MOV     RANGE_TOP,39*100    ; 8086,8MHz用の定数をセット
    MOV     RANGE_BOTTOM,35*100
    JMP     GET_RANGE_EXIT

;
GET_RANGE_2:
    MOV     RANGE_TOP,53*100    ; 8086,5MHz用の定数をセット
    MOV     RANGE_BOTTOM,49*100

;
GET_RANGE_EXIT:
    POP     ES
    RET
GET_RANGE ENDP

;
MESSAGE DB    13,10
        DB    'プログラムは一気に実行させて下さいね。'
        DB    13,10,'$'

;
ENDING  DB    13,10
        DB    'プログラムは正規に実行されました。'
        DB    13,10,'$'

;
BAD      DB    13,10
        DB    'あなたはプログラムを止めましたね?'
        DB    13,10,'$'

;
CHKEXE2 ENDP
;
CODE     ENDS
;
        END      CHKEXE2

```

■図 1.7 CHKEXE2.COM ダンプリスト

```

00000000 : EB 0F 90 00 00 00 00 00 00 00 00 00 80 0C F0 : 306
00000010 : 0A E8 B4 00 C7 06 08 01 00 00 E8 3C 00 8D 16 11 : 457
00000020 : 02 B4 09 CD 21 89 C8 00 51 33 C9 E2 FE 59 E2 F8 : 88E
00000030 : E8 5F 00 A1 0B 01 3B 06 0D 01 77 06 3B 06 0F 01 : 311
00000040 : 73 0A 8D 16 63 02 B4 09 CD 21 E8 FE 8D 16 3C 02 : 5FA
00000050 : B4 09 CD 21 88 00 4C CD 21 FB 1E 06 33 C0 8E D8 : 715
00000060 : C4 06 20 00 36 A3 07 01 36 8C 06 09 01 C4 06 1C : 383
00000070 : 00 36 A3 03 01 36 8C 06 05 01 C7 06 20 00 BA 01 : 353
00000080 : 8C 0E 22 00 07 B4 02 89 00 00 8D 1E C7 01 CD 1C : 48E
00000090 : 1F C3 FB 1E 06 E4 02 0C 01 E6 02 33 C0 8E D8 36 : 668
000000A0 : C4 06 03 01 A3 1C 00 8C 06 1E 00 36 C4 06 07 01 : 345
000000B0 : A3 20 00 8C 06 22 00 07 1F C3 FB 50 2E FF 06 0B : 4E9
000000C0 : 01 B0 20 E6 00 58 CF CF 06 33 C0 8E C0 26 A0 01 : 68B
000000D0 : 05 24 C0 3C 40 74 38 3C C0 75 11 2E C7 06 0D 01 : 49C
000000E0 : 3C 0F 2E C7 06 0F 01 10 0E EB 24 90 3C 80 75 11 : 455
000000F0 : 2E C7 06 0D 01 3C 0F 2E C7 06 0F 01 AC 0D EB 0F : 412
00000100 : 90 2E C7 06 0D 01 B4 14 2E C7 06 0F 01 24 13 07 : 3AA
00000110 : C3 0D 0A 83 76 83 8D 83 4F 83 89 83 80 82 CD 88 : 79B
00000120 : EA 8B 43 82 C9 8E C0 8D 73 82 B3 82 B9 82 C4 89 : 990
00000130 : BA 82 B3 82 A2 82 CB 81 44 0D 0A 24 0D 0A 83 76 : 670
00000140 : 83 8D 83 4F 83 89 83 80 82 CD 90 B3 8B 4B 82 C9 : 8A4
00000150 : 8E C0 8D 73 82 B3 82 EA 82 DC 82 B5 82 BD 81 44 : 988
00000160 : 0D 0A 24 0D 0A 82 A0 82 C8 82 BD 82 CD 83 76 83 : 6C8
00000170 : 8D 83 4F 83 89 83 80 82 F0 8E 7E 82 DF 82 DC 82 : 92D
00000180 : B5 82 BD 82 CB 81 48 0D 0A 24 : 445


```

■図 1.7 CHKEXE2.COM 実行例

A>CHKEXE2  ————ダイレクトに実行する

プログラムは一気に実行させて下さいね。

プログラムは正規に実行されました。——正常に終了する

A>SYMDEB B:\*\CMDS\CHKEXE2.COM  ————デバッガで動作させる

Microsoft Symbolic Debug Utility

Version 3.01

(C)Copyright Microsoft Corp 1984, 1985

Processor is [8086]

-U 

```

30C5:0100 EB0F          JMP     0111
30C5:0102 90             NOP
30C5:0103 0000          ADD     [BX+SI],AL
30C5:0105 0000          ADD     [BX+SI],AL
30C5:0107 0000          ADD     [BX+SI],AL
30C5:0109 0000          ADD     [BX+SI],AL
30C5:010B 0000          ADD     [BX+SI],AL
30C5:010D 800CF0        OR      Byte Ptr [SI],F0

```

-U 

```

30C5:0110 0AE8          OR      CH,AL
30C5:0112 B500          MOV     CH,00
30C5:0114 C7060B010000 MOV     Word Ptr [010B],0000
30C5:011A E83D00          CALL    015A
30C5:011D 8D161202      LEA     DX,[0212]
30C5:0121 B409          MOV     AH,09
30C5:0123 CD21          INT     21
30C5:0125 B9C800          MOV     CX,00C8

```

-G125  ————ループの手前まで実行させる

プログラムは一気に実行させて下さいね。

AX=0924 BX=01C8 CX=0000 DX=0212 SP=FFFE BP=0000 SI=0000 DI=0000  
DS=30C5 ES=30C5 SS=30C5 CS=30C5 IP=0125 NV UP EI PL ZR NA PE NC

```

30C5:0125 B9C800      MOV     CX,00C8
-U [A]
30C5:0128 51          PUSH    CX
30C5:0129 33C9        XOR     CX,CX
30C5:012B E2FE        LOOP   012B
30C5:012D 59          POP     CX
30C5:012E E2F8        LOOP   0128
30C5:0130 E86000      CALL   0193
30C5:0133 A10801      MOV     AX,[010B]
30C5:0136 3B060D01    CMP     AX,[010D]
-G [A]
AX=0924 BX=01C8 CX=0000 DX=0212 SP=FFFE BP=0000 SI=0000 DI=0000
DS=30C5 ES=30C5 SS=30C5 CS=30C5 IP=0130 NV UP EI PL ZR NA PE NC
30C5:0130 E86000      CALL   0193
-6 [A]

```

あなたはプログラムを止めたね?—時間がかかりすぎた

## ■タイムアウト割り込みを用いる

### ○考え方

インターバルタイマを利用するのですが、実際に時刻のカウントはせず、規定時間までに処理が終了できるかどうかを判別します。

### ○実行方法

インターバルタイマの機能には一定時間ごとの割り込み発生と、一定時間経過後の割り込み発生があります。ここでは、この一定時間経過後の割り込みを利用します。まず、何秒後にタイムアウトにするかを設定します。またプログラム中では、タイムアウトになるまでに、行うべき処理をすべて行ったかどうかを表すフラグを初期化し、行ったならばその時点でフラグをセットします。タイムアウト処理ルーチンではそのフラグを参照し、処理が未完であれば無限ループに陥ります。以上の手順を書き表せば、次のようになります。

.....

COMPLETE-FLAG      DB      0      ; 処理終了フラグ

```

.....

CALL    NIT- TIMER           ; タイマの初期化

.....           ; タイムアウト前に行うべき処理

MOV     COMPLETE-FLAG,-1    ; 処理終了フラグの
                           セット
.....

TIMEOUT  PROC    FAR        ; タイムアウト時の処理を行う関数

        PUSH    AX
        MOV     AX,COMPLETE-FLAG
        CMP     AX,-1
        JNE     $           ; 処理未終了ならば無限ループ
        POP     AX
        IRET
TIMEOUT  ENDP

.....

```

ここで、流れを特に説明する必要はないでしょうが、一応説明しておきますと、まず設けられている変数 **COMPLETE-FLAG** は、タイムアウト前に済ませておくべき処理を実行終了したらセットされるものです。また、プロシージャの内容が書かれていませんが、**INIT- TIMER** はタイマを初期化するプロシージャです。プロシージャ **TIME- OUT** では、セットされているはずの変数 **COMPLETE-FLAG** を参照し、セットされていなければ、処理が未終了として無限ループに陥ります。処理が終了していればそのまま戻ります。

## ○対処方法

基本的には先の3例と同様です。チェックを行っている箇所を捜し出して潰します。

## ○サンプル

インターバルタイマによって、処理がきちんと終了しているかどうかを調べて、そうでなければ無限ループに陥るプログラムCHKEXE3.COMと、その実行例を図1.8として示します。トレースを行ったりブレークポイントを置いて実行した場合には、間違いなく無限ループに陥りますので注意してください。

■図 1.8 CHKEXE3. ASM ソースリスト

```

:
: *****
:
:      CHKEXE3.ASM
:
:      SYMDEBの機能を無効にするサンプル(6)
:
:      このプログラムは、実行時間に制限を設け、タイムアウト割り込みによ
:      って実行が正規に行われているかチェックします。
:      注意！
:      ○実行が正規に行われていない場合、無限ループに陥ります。
:      ○CPUには80286を使用しないで下さい。
:      ○動作中にクロック周波数を切り替えないで下さい。
:
:      COPYRIGHT(C) 1987 BY SHUWA SYSTEM TRADING CO.,LTD.
:
:      LAST MODIFIED ON FEBRUARY 6TH,1987
:
: *****
CODE    SEGMENT
        ASSUME    CS:CODE,DS:CODE,ES:CODE,SS:CODE
:
:      ORG        100H
:
CHKEXE3 PROC
:
:      JMP        MAIN
:
:      EXEC_TIME   DW        32*100    ; 実行に要する時間
:      COMPLETE_FLAG DW        ?        ; 処理終了フラグ
:      CHECKED_FLAG  DW        ?        ; チェック終了フラグ
:
:      MAIN:
:      MOV        COMPLETE_FLAG,0    ; 処理終了フラグを初期化する
:      MOV        CHECKED_FLAG,0     ; チェック終了フラグを初期化する
:      CALL       GET_NEEDED          ; 実行に要する時間を得る
:      CALL       INIT_TIMER          ; インターバルタイマを初期化する

```

```

;
;      LEA      DX,MESSAGE      ; 警告を表示
;      MOV      AH,9
;      INT      21H
;
;      MOV      CX,200          ; ダミーループ
;
DUMMY_LOOP:
    PUSH      CX
    XOR       CX,CX
    LOOP      $
    POP       CX
    LOOP      DUMMY_LOOP
;
;      MOV      COMPLETE_FLAG,-1      ; 処理終了フラグのセット
;      LEA      DX,ENDING      ; おじに終了したことを告げるメッセージ
;      MOV      AH,9
;      INT      21H
;
WAIT_LOOP:
    MOV      AX,CHECKED_FLAG ; チェックは済んでいるかチェック
    CMP      AX,-1
    JNE      WAIT_LOOP
;
;      MOV      AX,4C00H          ; プログラム終了
;      INT      21H
;
INIT_TIMER:
    PROC      NEAR              ; インターバルタイマの初期化
    MOV      AH,2              ; インターバルタイマの起動
    MOV      CX,EXEC_TIME      ; タイムアウトまでの時間
    LEA      BX,TIMEOUT        ; タイムアウト時に呼ばれるルーチン
    INT      1CH
    RET
INIT_TIMER:
    ENDP
;
TIMEOUT:
    PROC      NEAR              ; タイムアウト時に呼ばれて実行終了をチェック
    PUSH     AX
    MOV      AX,CS:COMPLETE_FLAG ; 処理終了フラグを参照
    CMP      AX,-1
    JE       TIMEOUT_EXIT
;
TIMEOUT_LOOP:
    MOV      AH,17H            ; ブザーを小刻みにならす
    INT      18H
    XOR      CX,CX
    LOOP     $
    MOV      AH,18H
    INT      18H
    XOR      CX,CX
    LOOP     $
    JMP      TIMEOUT_LOOP
;
TIMEOUT_EXIT:
    MOV      CS:CHECKED_FLAG,AX ; チェック済フラグをセット
    POP      AX
    IRET
TIMEOUT:
    ENDP
;
GET_NEEDED:
    PROC      NEAR              ; 実行所要時間を算出
    PUSH     ES
    XOR      AX,AX
    MOV      ES,AX
    MOV      AL,ES:[501H]      ; システム情報を取り出す
    AND      AL,11000000B      ; クロック周波数・CPUを判別する
    CMP      AL,01000000B      ; V30,10MHzか?
    JE       GET_NEEDED_EXIT

```



```

;
;      CMP      AL,11000000B      ; V30,8MHzか?
;      JNE      GET_NEEDED_1
;
;      MOV      EXEC_TIME,39*100      ; V30,8MHz用の定数をセット
;      JMP      GET_NEEDED_EXIT
;
GET_NEEDED_1:
;      CMP      AL,10000000B      ; 8086,8MHzか?
;      JNE      GET_NEEDED_2
;
;      MOV      EXEC_TIME,39*100      ; 8086,8MHz用の定数をセット
;      JMP      GET_NEEDED_EXIT
;
GET_NEEDED_2:
;      MOV      EXEC_TIME,53*100      ; 8086,5MHz用の定数をセット
;
GET_NEEDED_EXIT:
;      POP      ES
;      RET
GET_NEEDED      ENDP
;
MESSAGE DB      13,10
;      DB      'プログラムは一気に実行させて下さいね。'
;      DB      13,10,'$'
;
ENDING DB      13,10
;      DB      'プログラムは正規に実行されました。'
;      DB      13,10,'$'
;
;
CHKEXE3 ENDP
;
CODE      ENDS
;
END      CHKEXE3

```

## ■図 1.8 CHKEXE3.COM ダンプリスト

```

00000000 : EB 07 90 80 0C 00 00 00 00 C7 06 05 01 00 00 C7 : 3A8
00000010 : 06 07 01 00 00 E8 60 00 E8 2E 00 8D 16 A9 01 B4 : 46D
00000020 : 09 CD 21 B9 C8 00 51 33 C9 E2 FE 59 E2 F8 C7 06 : 8A5
00000030 : 05 01 FF FF 8D 16 D4 01 B4 09 CD 21 A1 07 01 3D : 60D
00000040 : FF FF 75 F8 B8 00 4C CD 21 B4 02 8B 0E 03 01 8D : 73D
00000050 : 1E 56 01 CD 1C C3 50 2E A1 05 01 3D FF FF 74 12 : 607
00000060 : B4 17 CD 18 33 C9 E2 FE B4 18 CD 18 33 C9 E2 FE : 919
00000070 : EB EE 2E A3 07 01 58 CF 06 33 C0 8E C0 26 A0 01 : 6E7
00000080 : 05 24 C0 3C 40 74 20 3C C0 75 09 C7 06 03 01 3C : 480
00000090 : 0F EB 14 90 3C 80 75 09 C7 06 03 01 3C 0F EB 07 : 4E6
000000A0 : 90 C7 06 03 01 B4 14 07 C3 0D 0A 83 76 83 8D 83 : 596
000000B0 : 4F 83 89 83 80 82 CD 88 EA 8B 43 82 C9 8E C0 8D : 913
000000C0 : 73 82 B3 82 B9 82 C4 89 BA 82 B3 82 A2 82 CB 81 : 993
000000D0 : 44 0D 0A 24 0D 0A 83 76 83 8D 83 4F 83 89 83 80 : 580
000000E0 : 82 CD 90 B3 8B 48 82 C9 8E C0 8D 73 82 B3 82 EA : 9A2
000000F0 : 82 DC 82 B5 82 BD 81 44 0D 0A 24 : 4D4


```

## ■ 図 1.8 CHKEXE3.COM 実行例

A>CHKEXE3  —ダイレクトに実行

プログラムは一気に実行させて下さいね。

プログラムは正規に実行されました。—— 正常に終了する

A>SYMDEB B:¥CMDS¥CHKEXE3.COM  —デバッガで動作させる

Microsoft Symbolic Debug Utility

Version 3.01

(C)Copyright Microsoft Corp 1984, 1985

Processor is [8086]

-U 

```
2F79:0100 EB07      JMP     0109
2F79:0102 90        NOP
2F79:0103 800C00     OR      Byte Ptr [SI],00
2F79:0106 0000     ADD     [BX+SI],AL
2F79:0108 00C7     ADD     BH,AL
2F79:010A 06       PUSH    ES
2F79:010B 050100    ADD     AX,0001
2F79:010E 00C7     ADD     BH,AL
```

-U109 

```
2F79:0109 C70605010000    MOV     Word Ptr [0105],0000
2F79:010F C70607010000    MOV     Word Ptr [0107],0000
2F79:0115 E86000     CALL    0178
2F79:0118 E82E00     CALL    0149
2F79:011B 8D16A901    LEA     DX,[01A9]
2F79:011F B409     MOV     AH,09
2F79:0121 CD21     INT     21
2F79:0123 B9C800     MOV     CX,00C8
```

-G123 

ループの手前まで実行。この時点でタイマは動いている

プログラムは一気に実行させて下さいね。

AX=0924 BX=0156 CX=0C80 DX=01A9 SP=FFFE BP=0000 SI=0000 DI=0000

DS=2F79 ES=2F79 SS=2F79 CS=2F79 IP=0123 NV UP EI PL ZR NA PE NC

2F79:0123 B9C800 MOV CX,00C8

-U 

```
2F79:0126 51       PUSH    CX
2F79:0127 33C9     XOR     CX,CX
2F79:0129 E2FE     LOOP    0129
2F79:012B 59       POP     CX
2F79:012C E2F8     LOOP    0126
2F79:012E C7060501FFFF    MOV     Word Ptr [0105],FFFF
2F79:0134 8D16D401    LEA     DX,[01D4]
2F79:0138 B409     MOV     AH,09
```

-T100 

トレースしてみる

AX=0924 BX=0156 CX=00C8 DX=01A9 SP=FFFE BP=0000 SI=0000 DI=0000

DS=2F79 ES=2F79 SS=2F79 CS=2F79 IP=0126 NV UP EI PL ZR NA PE NC

2F79:0126 51 PUSH CX

AX=0924 BX=0156 CX=00C8 DX=01A9 SP=FFFC BP=0000 SI=0000 DI=0000

DS=2F79 ES=2F79 SS=2F79 CS=2F79 IP=0127 NV UP EI PL ZR NA PE NC

2F79:0127 33C9 XOR CX,CX

AX=0924 BX=0156 CX=0000 DX=01A9 SP=FFFC BP=0000 SI=0000 DI=0000

DS=2F79 ES=2F79 SS=2F79 CS=2F79 IP=0129 NV UP EI PL ZR NA PE NC

2F79:0129 E2FE LOOP 0129

^C 

ループを切る

-U 

```
2F79:012B 59       POP     CX
2F79:012C E2F8     LOOP    0126
2F79:012E C7060501FFFF    MOV     Word Ptr [0105],FFFF
2F79:0134 8D16D401    LEA     DX,[01D4]
2F79:0138 B409     MOV     AH,09
2F79:013A CD21     INT     21
2F79:013C A10701    MOV     AX,[0107]
2F79:013F 3DFFFF     CMP     AX,FFFF
```

-G 

一気に実行

このあたりでブザーが鳴ってしまう

## 1.4 常駐型ツールに対抗する

### ○考え方

SYMDEB は MS-DOS に準拠したツールであり、あまりにも簡単に封じられてしまうため、最近では、強力な解析機能を備えたツールが登場しています。その中には、MS-DOS とは関係のないメモリ空間に位置し、MS-DOS の管理を受けずに動作するものもあります。これらを封じるためのテクニックの一つを、ここで紹介します。

### ○実現方法

実現は簡単であり完璧とはいえませんが、自分の存在する領域以降や、VRAM など MS-DOS で直接使用しない領域をすべて破壊してしまう方法が考えられます。ここで重要なのは、システムの設定したメモリサイズなど信用せずに、すべて自分で調査して破壊に移ることです。

### ○サンプル

メモリの実装されている領域すべて（すでにシステムの存在している領域を除く）を破壊するプログラム CLRMEM.COM を、図 1.9 として示します。VRAM 等も破壊しますので表示は乱れて、グラフィック VRAM に RAM ディスクを構築していた場合には、内容が破壊されてしまいますので注意してください。

■ 図 1.9 CLRMEM.ASM ソースリスト

```

;*****
;
;      CLRMEM.ASM
;
;      システム領域以外のメモリをすべて破壊するサンプル
;
;      このプログラムは、MS-DOSや自らで使用されていない領域を、
;      VRAMを含めてすべて破壊するものです。よって、画面は乱れ、
;      RAMディスク等をGVRAM上に構築している場合には、すべて破壊され
;      ますので注意して下さい。
;
;      COPYRIGHT(C) 1987 BY SHUWA SYSTEM TRADING CO.,LTD.
;
;      LAST MODIFIED ON FEBRUARY 5TH,1987
;*****
CODE    SEGMENT
        ASSUME  CS:CODE,DS:CODE,ES:CODE,SS:CODE
;
;      ORG      100H
;
CLRMEM  PROC
;
;      LEA      SP,STACK      ; ローカルスタックの設定
;      LEA      DX,MESSAGE    ; 警告を表示
;      MOV      AH,9
;      INT      21H
;
;      MOV      AX,CS          ; 自らの終端を絶対アドレスへ変換
;      XOR      DX,DX
;      SAL      AX,1           ; 32ビットの左シフト
;      RCL      DX,1
;      SAL      AX,1
;      RCL      DX,1
;      SAL      AX,1
;      RCL      DX,1
;      SAL      AX,1
;      RCL      DX,1
;      ADD      AX,OFFSET STACK ; オフセットを加える
;      ADC      DX,0           ; 上位へ繰り上げる
;
;      FUSH     AX              ; オフセット部を待避
;      MOV      AX,DX           ; セグメント部の切り出し
;      XCHG     AL,AH
;      MOV      CL,4
;      SHL      AX,CL
;      MOV      ES,AX           ; ES = 未使用領域セグメント
;      MOV      DS,AX           ; DI = 未使用領域オフセット
;      POP      DI
;
;      XOR      CX,CX           ; 破壊するバイト数を計算
;      SUB      CX,DI
;      MOV      SI,DI
;      CLD
;
;      DESTROY1:
;      LODSB
;      NOT      AL              ; ビット反転をする破壊行為
;      STOSB
;      LOOP     DESTROY1
;
;      DESTROY2:
;      MOV      AX,ES           ; セグメントがメインRAMの終端かチェックする
;      CMP      AX,9000H
;      JE       DESTROY3       ; VRAMの破壊へ

```

```

; 次の物理セグメントへ
ADD     AX,1000H
MOV     ES,AX
MOV     DS,AX
XOR     SI,SI           ; 転送アドレスをセット
MOV     DI,SI
MOV     CX,SI          ; 転送バイト数をセット

;
DESTROY2_LOOP:
    LODSB
    ROL     AL,1        ; ビットをシフトする破壊行為
    STOSB
    LOOP    DESTROY2_LOOP

;
    JMP     DESTROY2    ; 次の物理セグメントへ

; テキストVRAMの破壊
DESTROY3:
    MOV     AX,0A000H   ; テキストVRAMのセグメント
    MOV     ES,AX
    MOV     DS,AX
    XOR     SI,SI       ; 転送アドレスをセット
    MOV     DI,SI
    MOV     CX,8000H/2  ; 転送ワード数をセット

;
DESTROY3_LOOP:
    LODSW
    XCHG    AL,AH       ; 上下を交換する破壊行為
    STOSW
    LOOP    DESTROY3_LOOP

;
    MOV     AL,1        ; 拡張GVRAMをアクティブにする
    OUT     6AH,AL

;
    XOR     AL,AL       ; バンク0から始める
    PUSH    AX

;
DESTROY4:
    OUT     0A6H,AL     ; グラフィックVRAMを破壊（青，赤）
    MOV     AX,0A800H   ; バンクを選択
    MOV     ES,AX       ; グラフィックVRAMのセグメント
    MOV     DS,AX
    XOR     SI,SI       ; 転送アドレスをセット
    MOV     DI,SI
    MOV     CX,8000H    ; 転送ワード数をセット

;
DESTROY4_LOOP:
    LODSW
    ROR     AL,1        ; ビットを回転する破壊行為
    STOSW
    LOOP    DESTROY4_LOOP

;
DESTROY5:
    MOV     AX,0B800H   ; グラフィックVRAMを破壊（緑）
    MOV     ES,AX       ; グラフィックVRAMのセグメント
    MOV     DS,AX
    XOR     SI,SI       ; 転送アドレスをセット
    MOV     DI,SI
    MOV     CX,4000H    ; 転送ワード数をセット

;
DESTROY5_LOOP:
    LODSW
    ROR     AL,1        ; ビットを回転する破壊行為
    STOSW
    LOOP    DESTROY5_LOOP

;
DESTROY6:
    MOV     AX,0E000H   ; グラフィックVRAMを破壊（拡張）
    MOV     ES,AX       ; グラフィックVRAMのセグメント
    MOV     DS,AX
    XOR     SI,SI       ; 転送アドレスをセット
    MOV     DI,SI
    MOV     CX,4000H    ; 転送ワード数をセット

```

```

MOV     ES,AX
MOV     DS,AX
XOR     SI,SI           ; 転送アドレスをセット
MOV     DI,SI
MOV     CX,4000H       ; 転送ワード数をセット
;
DESTROY6_LOOP:
LODSW
ROR     AX,1           ; ビットを回転する破壊行為
STOSW
LOOP    DESTROY6_LOOP
;
POP     AX
INC     AL             ; 次のバンクへ
CMP     AL,2           ; 0, 1 バンクを破壊したか?
JB      DESTROY4
;
MOV     AX,CS
MOV     DS,AX
LEA     DX,ENDING      ; 結果を表示
MOV     AH,9
INT     21H
;
MOV     AX,4C00H       ; プログラム終了
INT     21H
;
MESSAGE DB 13,10
DB 'VRAMを含めてメモリをクリアします。'
DB 13,10,'$'
;
ENDING  DB 13,10
DB 'メモリはクリアされました。'
DB 13,10,'$'
;
DW      128 DUP(?)     ; ローカルスタック
;
STACK   LABEL WORD     ; ローカルスタックポインタ
;
CLRMEM  ENDP
;
CODE    ENDS
;
END      CLRMEM

```

■図 1.9 CLRMEM.COM ダンプリスト

```

00000000 : 8D 26 14 03 8D 16 CE 01 84 09 CD 21 8C C8 33 D2 : 640
00000010 : D1 E0 D1 D2 D1 E0 D1 D2 D1 E0 D1 D2 D1 D2 : D50
00000020 : 05 14 03 83 D2 00 50 88 C2 86 C4 81 04 D3 E0 8E : 74E
00000030 : C0 8E D8 5F 33 C9 28 CF 88 F7 FC AC F6 D0 AA E2 : AF7
00000040 : FA 8C C0 3D 00 90 74 15 05 00 10 8E C0 8E D8 33 : 698
00000050 : F6 88 FE 88 CE AC D0 C0 AA E2 FA EB E4 B8 00 A0 : 8C1
00000060 : 8E C0 8E D8 33 F6 88 FE B9 00 40 AD 86 C4 A8 E2 : 9E3
00000070 : FA B0 01 E6 6A 32 C0 50 E6 A6 B8 00 A8 8E C0 8E : 905
00000080 : D8 33 F6 88 FE B9 00 80 AD D1 C8 A8 E2 FA B8 00 : A48
00000090 : B8 8E C0 8E D8 33 F6 88 FE B9 00 40 AD D1 C8 A8 : A08
000000A0 : E2 FA B8 00 E0 8E C0 8E D8 33 F6 88 FE B9 00 40 : 9D3
000000B0 : AD D1 C8 A8 E2 FA 58 FE C0 3C 02 72 B8 8C C8 8E : A30
000000C0 : D8 8D 16 F5 01 84 09 CD 21 B8 00 4C CD 21 0D 0A : 625
000000D0 : 56 52 41 4D 82 F0 8A DC 82 DF 82 C4 83 81 83 82 : 88E
000000E0 : 83 8A 82 F0 83 4E 83 8A 83 41 82 B5 82 DC 82 B7 : 8EF
000000F0 : 81 44 0D 0A 24 0D 0A 83 81 83 82 83 8A 82 CD 83 : 5FF
00000100 : 4E 83 8A 83 41 82 83 82 EA 82 DC 82 B5 82 BD 81 : 915
00000110 : 44 0D 0A 24 00 00 00 00 00 00 00 00 00 00 00 : 07F

```





# 2

## 暗号化のテクニック

暗号化は、プログラムを読みにくくし、リストどおりにプログラムが動作しないというもので、これは、あらゆるテクニックと組み合わせても効果的なものです。たとえば、プログラムを置く場所を工夫しても、複雑な処理を行わせても、それがプログラムとしてきちんと読めてしまうのならば、その効果も半減です。ここでは暗号化のためのテクニック、暗号化されていることをわからせないようにするテクニックについて紹介します。

## 2.1 暗号化の方法

暗号化とは一連の意味あるブロックについて、あるキーを用いてコードを変形し、それを、あるキーを用いて復活できるように加工するものです。暗号化の基本は元の形を留めずに、さらに元どおりに復元することができるということです。この変形の方法は重要です。ここでは、さまざまな暗号化のための方法について解説します。

### ■一定キーによる暗号化

#### ○考え方

復元性が高く、なおかつ、簡単に暗号化を施せる方法としては、一定キー・一定演算によるものがあげられます。



### ○実現方法

まず、暗号化のためのキーを決定します。ひとくちに一定キーといっても、その形式はいろいろと選ぶことができます。考えられるものとしては、以下のものがあげられるでしょう。

- ① 1バイトのデータ
- ② 2バイト（1ワード）のデータ
- ③ ブロックと同じ長さの文字列

①は、ブロックの1バイト1バイトを、キーを用いて変形していくものです。

②は、ブロックを2バイト（1ワード）ずつ、キーを用いて変形していくものです。

③は、ブロックの長さと同じ長さの文字列（キーの集合）を用いてブロック全体を変形するものです。

③については一定キーとはいいいくいかかもしれませんが、ブロック全体に対しては一定キーということができます。

キーが決まれば、変形の方法（これを演算といいます）を決定します。ここでは、変形の度合いと復元の可能性をうまくバランスさせなくてはなりません。このような点を考慮して、次のようなものがあげられます。

- ①' 反転 (NOT)
- ②' 排他的論理和 (exclusive OR)
- ③' 加算 (ADD)
- ④' 減算 (SUB)
- ⑤' 回転 (Rotate)

他のものでは、復元の可能性を考えた場合、あまり適当なものは存在しません。とりあえずはここにあげたものか、もしくはこれら

を組合せたもので十分でしょう。

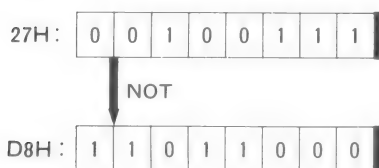
変形の方法に対して、復元の方法も決められなければなりません。

①'から⑤'に対応するものとして、以下のものがあげられます。

- ①'' 反転 (同じ)
- ②'' 排他的論理和 (同じ)
- ③'' 減算 (逆)
- ④'' 加算 (逆)
- ⑤'' 回転 (逆方向のもの)

①'と①''の組み合わせについては問題ないでしょう。反転された内容を再び反転するのですから、元に戻るのは容易に理解できるでしょう (図 2.1 参照)。

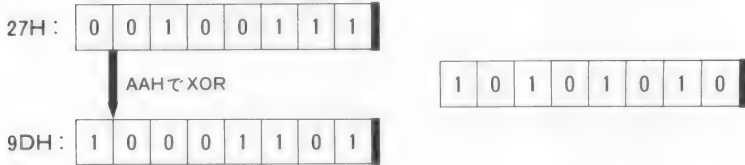
■図 2.1 反転の例



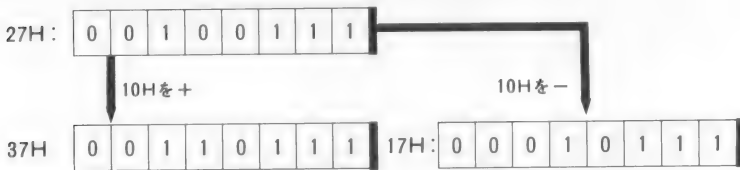
②'と②''の組み合わせは、部分的な反転とみなすことができます。このときキーの全ビットを 1 とすれば、反転と同じ効果を持ちます (図 2.2 参照)。

③'と③''の組み合わせと、④'と④''の組み合わせは、互いに逆のことを行っているといえます。すなわち変形時に加えた(減らした)分を、復元時に減らして(増やして)やるわけです (図 2.3 参照)。

■図 2.2 排他的論理和の例



■図 2.3 加算と減算の例



⑤'と⑤''は、回したビット列を逆方向に回すのですから元に戻るのは当然です（図 2.4 参照）。この場合、回転数がキーとしての値となるでしょう。

■図 2.4 回転の例



## ○対処方法

暗号化の場合、とにかく復元を行っている箇所は必ずあるはずですので、そこをまず見つけます。また、途中でどうも不自然な命令列があることに気付いたら、そこも暗号化が施されている可能性があります。特に、CALL 命令や JMP 命令において、飛び先がプログラムとして不自然な場合、暗号化が施されており、気付かないうちに復元が行われていると思っても間違いはないでしょう。ただし、わざと不自然な流れとしている場合もありますので（応用編 I・2 参照）、その点は留意しておいてください。

また復元している箇所を見つけたら、とにかく復元先を復元してみることです。そうでないと、暗号化されている部分を正しく解読することができません。復元のための方法については、2.2 に後述します。

## ○サンプル

SYMDEB によるオペレーションですが、このオペレーションは、オフセット 1000H に置かれているプログラムを、①と①'の組み合わせで暗号化するものです（この場合、キーは値として存在せず、サイズと演算方法のみが存在する）。元のプログラムと暗号化が施されたプログラムを比較してください（図 3.6 参照）。

■図 2.5 暗号化の例（1）

```

A>SYMDEB
Microsoft Symbolic Debug Utility
Version 3.01
(C) Copyright Microsoft Corp 1984, 1985
Processor is [8086]
~A1000
30B9:1000 MOV AH,9
30B9:1002 MOV DX,1000
30B9:1005 INT 21
30B9:1007 MOV AH,A
30B9:1009 MOV DX,1200
30B9:100C INT 21

```

暗号化されるプログラムを作成

メッセージを出力し、1行入力を行うプログラム

```

30B9:100E CMP BYTE [1201],0
30B9:1013 JE 1000
30B9:1015 RET
30B9:1016
-U1000 [a] ----- 暗号化するプログラムを作成
30B9:0100 MOV SI,1000
30B9:0103 MOV DI,1000
30B9:0106 MOV CX,20
30B9:0109 CLD
30B9:010A LODSB
30B9:010B NOT AL ----- 反転処理を施す
30B9:010D STOSB
30B9:010E LOOP 10A
30B9:0110
-U1000 [a] ----- もとのプログラムを確認
30B9:1000 B409 MOV AH,09
30B9:1002 BA0010 MOV DX,1000
30B9:1005 CD21 INT 21
30B9:1007 B40A MOV AH,0A
30B9:1009 BA0012 MOV DX,1200
30B9:100C CD21 INT 21
30B9:100E 803E011200 CMP Byte Ptr [1201],00
30B9:1013 74EB JZ 1000
-U [a]
30B9:1015 C3 RET
30B9:1016 OCC7 OR AL,C7
30B9:1018 06 PUSH ES
30B9:1019 E50B IN AX,0B
30B9:101B 8802 MOV [BP+SI],AL
30B9:101D 8C1EE70B MOV [0BE7],DS
30B9:1021 803E2E14FF CMP Byte Ptr [142E],FF
30B9:1026 75BD JNZ 0FE5
-G=100,110 [a] ----- 暗号化
AX=0018 BX=0000 CX=0000 DX=0000 SP=CE36 BP=0000 SI=1020 DI=1020
DS=30B9 ES=30B9 SS=30B9 CS=30B9 IP=0110 NV UP EI PL NZ NA PO NC
30B9:0110 CD21 INT 21 :Terminate Program
-U1000 [a] ----- 結果をみる
30B9:1000 4B DEC BX
30B9:1001 F645FFEF TEST Byte Ptr [DI-01],EF
30B9:1005 32DE XOR BL,DH
30B9:1007 4B DEC BX ----- さっぱりわからない
30B9:1008 F5 CMC
30B9:1009 45 INC BP
30B9:100A FFED JMP FAR BP
30B9:100C 32DE XOR BL,DH
-G=100,110 [a]
AX=00E7 BX=0000 CX=0000 DX=0000 SP=CE36 BP=0000 SI=1020 DI=1020
DS=30B9 ES=30B9 SS=30B9 CS=30B9 IP=0110 NV UP EI PL NZ NA PO NC
30B9:0110 CD21 INT 21 :Terminate Program
-U1000 [a] ----- 元に戻す
30B9:1000 B409 MOV AH,09
30B9:1002 BA0010 MOV DX,1000
30B9:1005 CD21 INT 21
30B9:1007 B40A MOV AH,0A
30B9:1009 BA0012 MOV DX,1200 ----- きちんと戻っている
30B9:100C CD21 INT 21
30B9:100E 803E011200 CMP Byte Ptr [1201],00
30B9:1013 74EB JZ 1000

```

## ■可変のキーを用いた暗号化

### ○考え方

一定キーを用いた場合、暗号化されていることがすぐにわかってしまったり、すぐに復元されてしまうという欠点があります。そこで可変のキーを用い、暗号化を複雑化するとともに、復元を困難にしようというものです。

### ○実現方法

可変のキーとひとくちにいっても、そのキーの決定については、一定キーの場合と同様に何通りもの方法があります。しかし復元できることが第一条件ですから、いいかげんなものを選ぶことはできません。そこで、考えられるのは以下のようなものです。

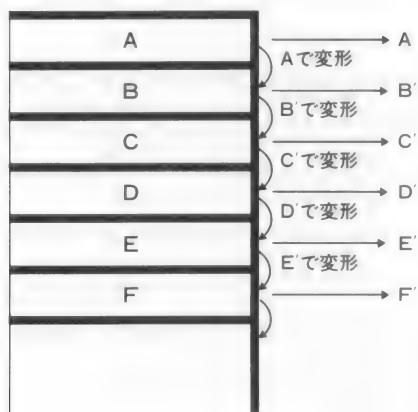
- ① 直前のデータを用いる
- ② アドレス値を用いる
- ③ 疑似乱数を用いる

①はあるブロック内の1バイト1バイト（2バイト以上でも同様）についてブロック全体のキーを定め、そのキーが1番目のデータを変形し、1番目のデータが2番目のデータを変形するという動作をブロック全体について行うものです（図2.6参照）。

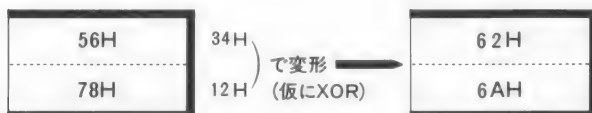
②は、そのデータが置かれているアドレスを用いて暗号化を施すものです。変形時と復元時でアドレスは変化しないので、キーとして有効なわけです（図2.7参照）。

③は、出現パターンが等しい疑似乱数を用いて暗号化を施すものです。疑似乱数発生のパターンを見破らない限り解読は困難です。

■図 2.6 直前のデータを用いた暗号化



■図 2.7 アドレス値を用いた暗号化



## ○対処方法

一定キーの場合と同様に、復元を行っている箇所を捜します。

## ○サンプル

例を図 2.8 として、SYMDEB によるオペレーションで示しておきますので参考にしてください。この場合、①と②の両方で暗号化を行っていますが、後者の場合、プログラムを置く位置によって結果が異なることに注意してください。

■図 2.8 暗号化の例 (2)

```

A>SYMDEB
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086]

-A1000
2F6D:1000 MOV AH,9
2F6D:1002 MOV DX,1100
2F6D:1005 INT 21
2F6D:1007 MOV DX,1200
2F6D:100A MOV AH,A
2F6D:100C INT 21
2F6D:100E CMP BYTE [1201],0
2F6D:1013 JE 1000
2F6D:1015 RET
2F6D:1016

-M1000 1015 2000
-U1000
2F6D:1000 B409 MOV AH,09
2F6D:1002 BA0011 MOV DX,1100
2F6D:1005 CD21 INT 21
2F6D:1007 BA0012 MOV DX,1200
2F6D:100A B40A MOV AH,0A
2F6D:100C CD21 INT 21
2F6D:100E 803E011200 CMP Byte Ptr [1201],00
2F6D:1013 74EB JZ 1000

-U2000
2F6D:2000 B409 MOV AH,09
2F6D:2002 BA0011 MOV DX,1100
2F6D:2005 CD21 INT 21
2F6D:2007 BA0012 MOV DX,1200
2F6D:200A B40A MOV AH,0A
2F6D:200C CD21 INT 21
2F6D:200E 803E011200 CMP Byte Ptr [1201],00
2F6D:2013 74EB JZ 2000

-A100
2F6D:0100 MOV SI,1000
2F6D:0103 MOV DI,1000
2F6D:0106 MOV CX,1F
2F6D:0109 MOV DL,AA
2F6D:010B CLD
2F6D:010C LODSB
2F6D:010D XOR AL,DL
2F6D:010F MOV DL,AL
2F6D:0111 STOSB
2F6D:0112 LOOP 10C
2F6D:0114

-G=100,114
AX=0060 BX=0000 CX=0000 DX=0060 SP=CF82 BP=0000 SI=101F DI=101F
DS=2F6D ES=2F6D SS=2F6D CS=2F6D IP=0114 NV UP EI PL NZ NA PE NC
2F6D:0114 0000 ADD [BX+SI],AL DS:101F=E7

-U1000
2F6D:1000 1E PUSH DS
2F6D:1001 17 POP SS
2F6D:1002 AD LODSW
2F6D:1003 AD LODSW
2F6D:1004 BC7150 MOV SP,5071
2F6D:1007 EAEAF84C46 JMP 464C:F8EA
2F6D:100C 8BA2A14 MOV BP,[BP+SI+142A]
2F6D:1010 150707 ADC AX,0707

-M2000 201F 1000
-A100
2F6D:0100 MOV SI,1000
2F6D:0103 MOV DI,1000
2F6D:0106 MOV CX,10*2
2F6D:0109 CLD

```

暗号化されるプログラムを作成

同様のものを異なるアドレスへ作成

確認

直前の結果を用いた暗号化

初期値はAAH

暗号化

結果をみる

さっぱりわからない

プログラムを元に戻す

アドレスを用いた暗号化



```

2F6D:010A MOV AX,SI
2F6D:010C OR AL,AH
2F6D:010E MOV DL,AL
2F6D:0110 LODSB
2F6D:0111 XOR AL,DL
2F6D:0113 STOSB
2F6D:0114 LOOP 10A
2F6D:0116
-G=100,116 [a] -----実行
AX=1076 BX=0000 CX=0000 DX=001F SP=CF82 BP=0000 SI=1020 DI=1020
DS=2F6D ES=2F6D SS=2F6D CS=2F6D IP=0116 NV UP EI PL NZ NA PO NC
2F6D:0116 0000 ADD [BX+SI],AL DS:1020=0B
-U1000 [a] -----結果をみる
2F6D:1000 A4 MOVSB
2F6D:1001 18A81305 SBB [BX+SI+0513],CH
2F6D:1005 D837 FDIV DWord Ptr [BX]
2F6D:1007 AD LODSW
2F6D:1008 180B SBB [BP+DI],CL
2F6D:100A AE SCASB
2F6D:100B 11D1 ADC CX,DX
2F6D:100D 3C9E CMP AL,9E
-A100 [a] -----アドレスを変更してみる
2F6D:0100 MOV SI,2000
2F6D:0103 MOV DI,2000
2F6D:0106
-G=100,116 [a] -----実行
AX=2056 BX=0000 CX=0000 DX=003F SP=CF82 BP=0000 SI=2020 DI=2020
DS=2F6D ES=2F6D SS=2F6D CS=2F6D IP=0116 NV UP EI PL NZ NA PE NC
2F6D:0116 0000 ADD [BX+SI],AL DS:2020=00
-U2000 [a] -----結果をみる
2F6D:2000 94 XCHG AX,SP
2F6D:2001 28982335 SUB [BX+SI+3523],BL
2F6D:2005 E8079D CALL BDOF
2F6D:2008 283B SUB [BP+DI],BH
2F6D:200A 9E SAHF -----結果が異なる
2F6D:200B 21E1 AND CX,SP
2F6D:200D 0CAE OR AL,AE
2F6D:200F 1131 ADC [BX+DI],SI
-

```

## ■キーを明示しない暗号化

### ○考え方

キーを具体的な値として示さず、どこか別の箇所からキーが得られるというものです。効果的に使用するには、キーを解読によって得にくいものにするといでしょう。

### ○実現方法

キーの決定には、次の手段を用いることができます。

- ① 計算による
- ② リターンコードなどを用いる

①では、複雑な演算を連続して行い、結果として得られたデータをキーとするものです。キーの算出が面倒であればあるほど、この方法は効果を増します。

②は、ディスク BIOS などのリターンコード（処理の終了状況を表したデータ）をキーとするものです。リターンコードとしてどのような値が返されるかわからなければ、キーも知ることができないのですから、ディスク BIOS の処理が理解できない場合、または結果が予測できない場合には非常に効果的です。この方法は 3, 4 年前にあるワープロソフトで用いられていました。ディスク上に正常な ID が存在しない場合に、返されるコードを知ることができなければ復元はできないのです。

### ○対処方法

①については、とにかく計算を行い、得られるべきキーを求めることです。

②については、ディスク BIOS やディスクのフォーマットについての知識が必要なわけですから、それを知らなければなりません。

両者とも面倒なら、実行可能な場合に実行させてみて、キーを決定する時点でのキーの内容を得ておくことです。

### ○サンプル

割り込みベクタテーブル全体を 1 バイトずつ加算し、それをキーとして暗号化を施すものです。当然ながら割り込みベクタの状態が異なるシステムでは暗号化の結果が異なり、かつ復元はできません（図 2.9 参照）。

## ■図 2.9 暗号化の例 (3)

```

A>SYMDEB[2]
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086]

-A1000[2] ----- 暗号化されるプログラムを作成
4867:1000 MOV AH,9
4867:1002 MOV DX,1100
4867:1005 INT 21
4867:1007 MOV AH,A
4867:1009 MOV DX,1200
4867:100C INT 21
4867:100E CMP BYTE [1201],0
4867:1013 JE 1000
4867:1015 RET
4867:1016

-A100[2] ----- 暗号化するプログラムを作成
4867:0100 XOR AX,AX
4867:0102 MOV DS,AX
4867:0104 MOV SI,AX
4867:0106 MOV DX,AX
4867:0108 MOV CX,400
4867:010B LODSB
4867:010C ADD DX,AX
4867:010E LOOP 10B
4867:0110 XOR DL,DH
4867:0112 PUSH CS
4867:0113 POP DS
4867:0114 MOV SI,1000
4867:0117 MOV DI,1000
4867:011A MOV CX,20
4867:011D LODSB
4867:011E XOR AL,DL
4867:0120 STOSB
4867:0121 LOOP 11D
4867:0123

-G=100,123[2] ----- 暗号化
AX=009C BX=0000 CX=0000 DX=419C SP=FFFE BP=0000 SI=1020 DI=1020
DS=4867 ES=4867 SS=4867 CS=4867 IP=0123 NV UP EI NG NZ NA PE NC
4867:0123 0000 ADD [BX+SI],AL DS:1020=00
-U1000[2] ----- 結果をみる
4867:1000 2895269C SUB [DI+9C26],DL
4867:1004 8D51BD LEA DX,[BX+DI-43]
4867:1007 2896269C SUB [BP+9C26],DL
4867:100B 8E51BD MOV SS,[BX+DI-43] ----- さっぱりわからない
4867:100E 1CA2 SBB AL,A2
4867:1010 9D POPF
4867:1011 8E9CE877 MOV DS,[SI+77E8]
4867:1015 5F POP DI
-G=100,123[2] ----- 元に戻す
AX=0000 BX=0000 CX=0000 DX=419C SP=FFFE BP=0000 SI=1020 DI=1020
DS=4867 ES=4867 SS=4867 CS=4867 IP=0123 NV UP EI PL ZR NA PE NC
4867:0123 0000 ADD [BX+SI],AL DS:1020=00
-U1000[2] ----- 元に戻ったかみる
4867:1000 8409 MOV AH,09
4867:1002 BA0011 MOV DX,1100
4867:1005 CD21 INT 21
4867:1007 B40A MOV AH,0A ----- 戻った
4867:1009 BA0012 MOV DX,1200
4867:100C CD21 INT 21
4867:100E 803E011200 CMP Byte Ptr [1201],00
4867:1013 74EB JZ 1000
-G=100,123[2] ----- 再び暗号化
AX=009C BX=0000 CX=0000 DX=419C SP=FFFE BP=0000 SI=1020 DI=1020
DS=4867 ES=4867 SS=4867 CS=4867 IP=0123 NV UP EI NG NZ NA PE NC
4867:0123 0000 ADD [BX+SI],AL DS:1020=00
-E 0:3FF[2] ----- ベクタの一部を故意に書き換える

```

```

0000:03FF 06.00(Ⓜ)
-G=100,123(Ⓜ)
AX=000A BX=0000 CX=0000 DX=4196 SP=FFFE BP=0000 SI=1020 DI=1020
DS=4867 ES=4867 SS=4867 CS=4867 IP=0123 NV UP EI PL NZ NA PE NC
4867:0123 0000 ADD [BX+SI],AL DS:1020=00
-U1000(Ⓜ)
4867:1000 BE0380 MOV SI,B003
4867:1003 0A1B OR BL,[BP+DI]
4867:1005 C728BE00 MOV Word Ptr [BP+DI],00BE
4867:1009 B00A MOV AL,0A
4867:100B 18C7 SBB BH,AL
4867:100D 2B8A340B SUB CX,[BP+SI+0B34]
4867:1011 180A SBB [BP+SI],CL
4867:1013 7EE1 JLE OFF6

```

## ■並べ換えによる暗号化

### ○考え方

キーを用いた変形は、常に復元の可能性と隣り合わせになるため、あまり複雑なことを行うと復元できなくなりますが、これは、キーを用いずにブロック内に操作を施して、結果的に暗号化としてしまうというものです。いうなれば“パズル16”のようなもので、目茶苦茶に並べ換えられたブロック内を、一定の手順に従って元に戻せば復元も容易です。

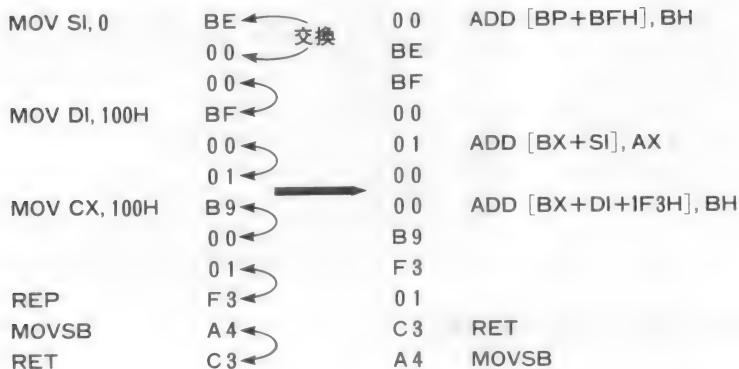
### ○実現方法

並べ換えの方法にはいくつかあげられます。

- ① 交換
- ② 回転
- ③ 反転
- ④ ランダム

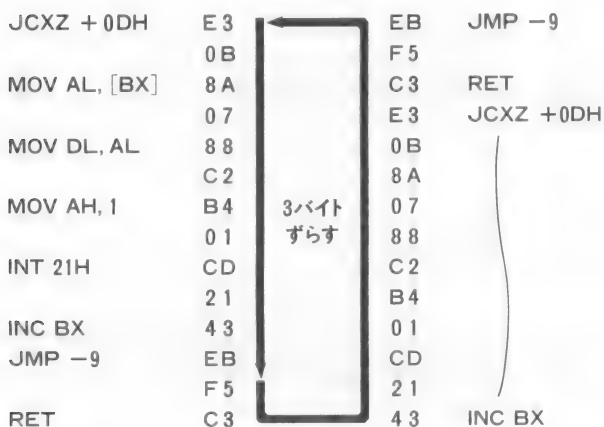
①は、ブロック内をたとえば1バイトずつ、隣り合う1バイトと交換するものです。これだけで8086の命令はまったく狂います(図2.10参照)。

■図 2.10 交換の例



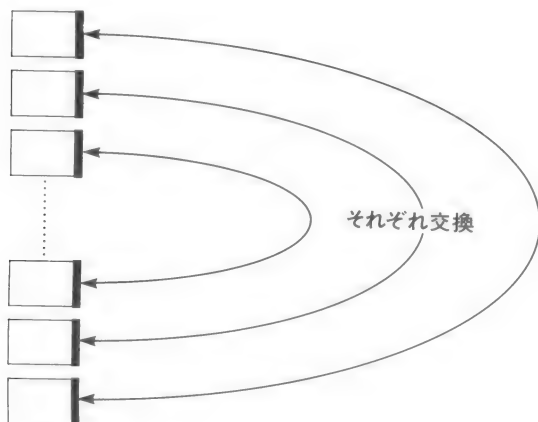
②は、ブロック内全体を回転するものです。金庫のダイヤルのように、正方向と逆方向の回転を何回か行って、複雑化するのもよいでしょう（図 2.11 参照）。

■図 2.11 回転の例



③は、ブロック内の低位と上位を反転するものです。いわば上下逆さまです（図 2.12 参照）。

■図 2.12 反転の例



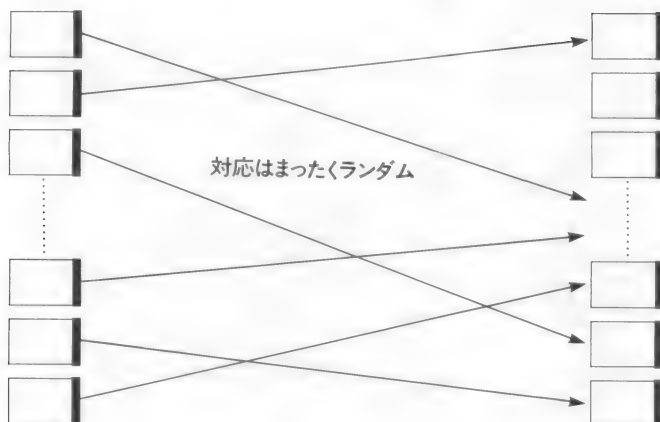
④は、ブロック内をある関数を定めて、一見ランダムに並べ換えるものです。復元時には変形時に用いた関数の逆関数を用います（図 2.13 参照）。

この場合、対応が 1 対 1 でないと復元することができなくなる可能性もありますから、関数の仕様には注意しなければなりません。なお、疑似乱数を用いるという方法もあります。これは BASIC の P オプションセーブで用いています。

### ○対処方法

とにかく復元を行っている箇所を捜し、並べ換えの方法を知るしかありません。

■図 2.13 ランダムの例



## ○サンプル

上下反転の例を図 2.14 として示します。SYMDEB によるオペレーションです。

■図 2.14 暗号化の例 (4)

```

A>SYMDEB
Microsoft Symbolic Debug Utility
Version 3.01
(C) Copyright Microsoft Corp 1984, 1985
Processor is [8086]
-A1000                                暗号化されるプログラムを入力
7B8B:1000 MOV AH,9
7B8B:1002 MOV DX,1100
7B8B:1005 INT 21
7B8B:1007 MOV AH,A
7B8B:1009 MOV DX,1200
7B8B:100C INT 21
7B8B:100E MOV AL,[1201]
7B8B:1011 OR AL,AL
7B8B:1013 JNE 1000
7B8B:1015 RET
7B8B:1016
-A100                                  暗号化するプログラムを入力
7B8B:0100 MOV SI,1000
7B8B:0103 MOV DI,1015
7B8B:0106 MOV CX,DI
7B8B:0108 SUB CX,SI
7B8B:010A INC CX
7B8B:010B SHR CX,1
7B8B:010D MOV AL,[SI]
7B8B:010F XCHG AL,[DI]
  
```

```

788B:0111 MOV [SI],AL
788B:0113 INC SI
788B:0114 DEC DI
788B:0115 LOOP 10D
788B:0117
-G=100,117 [?] 暗号化
AX=0012 BX=0000 CX=0000 DX=0000 SP=FFFE BP=0000 SI=100B DI=100A
DS=788B ES=788B SS=788B CS=788B IP=0117 NV UP EI PL NZ NA PE NC
788B:0117 0000 ADD [BX+SI],AL DS:100B=00
-U1000 [?] 結果をみる
788B:1000 C3 RET RET 命令が先頭にきている
788B:1001 EB75 JMP 1078
788B:1003 C00812 ROR Byte Ptr [BX+SI],12
788B:1006 01A021CD ADD [BX+SI+CD21],SP
788B:100A 1200 ADC AL,[BX+SI]
788B:100C BA0AB4 MOV DX,B40A
788B:100F 21CD AND BP,CX
788B:1011 1100 ADC [BX+SI],AX
-U [?]
788B:1013 BA09B4 MOV DX,B409
788B:1016 0000 ADD [BX+SI],AL
788B:1018 0000 ADD [BX+SI],AL
788B:101A 0000 ADD [BX+SI],AL
788B:101C 0000 ADD [BX+SI],AL
788B:101E 0000 ADD [BX+SI],AL
788B:1020 0000 ADD [BX+SI],AL
788B:1022 0000 ADD [BX+SI],AL
-G=100,117 [?] 元に戻す
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFFE BP=0000 SI=100B DI=100A
DS=788B ES=788B SS=788B CS=788B IP=0117 NV UP EI PL NZ NA PE NC
788B:0117 0000 ADD [BX+SI],AL DS:100B=12
-U1000 [?] 結果をみる
788B:1000 B409 MOV AH,09
788B:1002 BA0011 MOV DX,1100
788B:1005 CD21 INT 21
788B:1007 B40A MOV AH,0A
788B:1009 BA0012 MOV DX,1200
788B:100C CD21 INT 21
788B:100E A00112 MOV AL,[1201]
788B:1011 08C0 OR AL,AL
-U [?]
788B:1013 75EB JNZ 1000
788B:1015 C3 RET
788B:1016 0000 ADD [BX+SI],AL
788B:1018 0000 ADD [BX+SI],AL
788B:101A 0000 ADD [BX+SI],AL
788B:101C 0000 ADD [BX+SI],AL
788B:101E 0000 ADD [BX+SI],AL
788B:1020 0000 ADD [BX+SI],AL
-

```

## ■コマンドにより動作させる暗号化

### ○考え方

命令の実行が直接の効果を持つのではなく、それは単にコマンドを実行させる手順にすぎないというものです。この場合、コマンドの意味がわからなくては、結局、何が行われているのかわからない



のですから、復元の方法もない（必要ない）暗号化です。

### ○実現方法

まず、コマンドによって何を行わせるか、何が行えるのかということを決めます。単純にあげてみれば、

メモリアクセス  
各種計算  
ディスクアクセス

などが考えられるでしょうが、これらに対応したコマンドを決定し、それらを並べたコマンド表を実際のプログラムとするのです。似たような考え方は、ある有名なチェックルーチンが採用しています。

たとえば、

**メモリリード：識別コード"00H**

アドレス上位

アドレス下位

格納レジスタ（メモリ上に仮想レジスタを構成）

**加算：識別コード"10H**

被加数格納レジスタ

加数格納レジスタ

和格納レジスタ

といった具合で、これとは別に解釈し実行するプログラムが存在するのです。雰囲気としては、BASIC インタプリタに似ていると思います。

## 2.2 復元の方法

ここでは、すでに暗号化が施されているブロックについて、それを復元する方法を書きます。

プログラム解読時において、暗号化されている部分を復元するには以下の方法が考えられます。

- ① 復元を行っている部分を解読し、手動で復元する
- ② 復元を行っている部分を部分的に実行させ、復元する

①は、プログラムが実行できないときに行うか、とりあえず解読のみで復元してみようというときに試みられる方法です。復元のためのアルゴリズムを、とにかく知らなければなりませんので、手間のかかるのが欠点です。

②は、プログラムを部分的に走らせてみることでできる環境があり、実行させてみても支障のない場合です（支障のある場合については後述）。アルゴリズムを知るための解読は必要なく、また、確実に復元された内容を得られますので（手動で行うとミスの入り込む確率が高い）手軽といえます。しかしこの場合、復元させてみてもあとの動作に支障の出る場合があります、安全とはいえません。

さて、復元を手動で行う場合については特に細かな説明は加えず、ここでは、復元をプログラム自身によって行わせる場合について、説明してみましょう。

まず、復元を自動的に行わせる場合、その方法について明らかにするとします。たとえば、レジスタの値には何が必要で、どこからどこまでを実行させればよいかといった情報です。では実行させて

みましょう。みなさんも適当な状況を頭に描き、想像で実行させてください。実行が終り、暗号化の施されていた部分が正常になって、まともな命令の並びになっています。この場合は、うまくいったとみてよいでしょう。正常な部分のリストをプリンタへ打ち出すなりして保存しておくといよいでしょう。

ところで、このとき問題となるのはプログラムを部分的に実行させるということです。ご存じのとおり、SYMDEB などのデバグガを用いてプログラムの部分実行を行わせる場合、実行終了位置には必ずブレークポイント (INT3 命令) が置かれます。ここで気を付けていただきたいのは、このブレークポイントの存在が、復元の支障になるかもしれないということです。たとえば、以下のような場合があげられます。

## ■復元ルーチンのチェックサムをとっている場合

復元ルーチンが、自らのチェックサムをとりながら実行している場合、ブレークポイントに置かれた命令コードが、チェックサムを崩してしまう可能性があるからです。このような場合は、復元ルーチンのコピーをどこかの空き領域に作成し、それを代りに実行させるしかありません。

# 3

## プログラムをかくす

ファイルのかたちで存在するプログラムは、ふつう RAM 上にロードされて実行されますが、OS の規則に従った場合には、あまりにももともとに実行されてしまいます。プログラムの解読を困難にするためには、プログラムの配置場所に工夫をこらすのも、一つのテクニックです。

## 3.1 プログラムをVRAM上に置く

### ■プログラムをテキストVRAM上に置く

#### ○考え方

通常テキスト VRAM はメッセージの表示に用いる領域ですが、RAM であることには変りがないため、プログラムを置く場所としても使用することができます。解読に際しては、単なるメッセージの表示と思われることも多く、うまく使えば、たとえば暗号化と組み合わせればなかなか効果的です。

#### ○実現方法

よく行われるのは、A1000H からの領域を用いる方法です。この領域は、テキスト VRAM の 2 ページ目として用意されているもので、BASIC や MS-DOS からは使用されていません。そこで、

4KB という小さな領域ですが、プログラムを置く領域として十分に用いることができます。ただし、漢字 ROM を実装していない機種の場合には、メモリが不連続になるので注意してください。

この場合は、VRAM を単なる一時的な置き場所と考えていましたが、置き方を考えれば十分にプロテクトとして通用します。たとえばプログラムを PRINT 文 (BASIC) や printf() 関数 (C 言語) などを用いてテキスト画面に表示し、そこへ実行を移すものです。ただし、画面への表示において制御コードとみなされることによる (画面に表示される) プログラムの乱れや、テキストの表示方式については、十分に気を付けなければなりません。参考になるかもしれませんが、まずテキストの表示について触れておきましょう。

PC-9801 ではテキスト VRAM の形式は、図 3.1 のようになっています。

■図 3.1 テキスト VRAM の形式

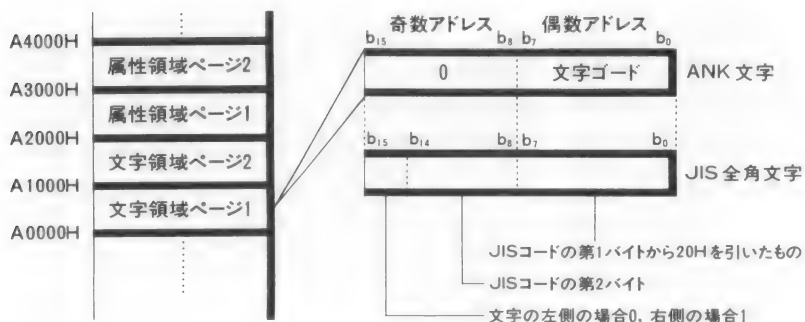


図 3.1 からわかるように、通常のキャラクタ（ANK 文字）の表示にも必ず 2 バイトの領域を要します。この場合、第 2 バイト目（奇数バイト）には必ず 00H が入って ANK 文字であることを表し、第 1 バイト目（偶数バイト）にはキャラクタコードが入ります。

また、日本語（JIS 全角文字）の表示には、4 バイトの領域を要します。この場合、第 1 バイト、第 2 バイト目が文字の左半分を受け持ち、第 3 バイト、第 4 バイト目が文字の右半分を受け持ちます。このとき、これらのデータは表示する文字のコードによって、以下のような規則を守らなければなりません。

#### [左半分の表示の場合]

第 1 バイト目……表示する文字コードの上位バイトから 2 0 H を引いた値

第 2 バイト目……表示する文字コードの下位バイト

#### [右半分の表示の場合]

第 3 バイト目……表示する文字コードの上位バイトから 2 0 H を引いた値

第 4 バイト目……表示する文字コードの下位バイトの最上位ビットを 1 とした値

さらに、JIS 半角文字の表示の場合には、2 バイトの領域で表示することができますが、表示するコードによって格納するデータが異なりその分類も面倒なので、ここでは詳しく説明しません。

例を示しますと、ANK 文字のスペース（20H）を表示する場合には、VRAM 上には 20H 00H の順にコードが入ります。また、JIS 全角文字の“亜”（JIS コード 3021H）を表示するには、10H, 21H, 10H, A1H が順に入ります。

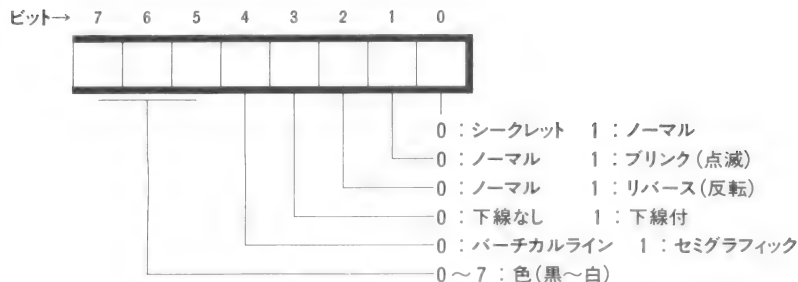
以上をまとめれば、テキスト VRAM 上に正しいプログラムを置くには、この変換規則を十分に理解しておかなければなりません。これはなかば強制的なものですから、置くことのできるプログラムにもかなりの制限が加えられます。もっとも工夫次第で何とかなるものです。

なお忘れてはならないことに、テキスト VRAM 上のプログラムを表示させてはいけないことがあります。せっかく特殊な位置にプログラムを置いたのですから、また当然、正常な文字列とはなりにくく目立ってしまいますから、それをかくさなければなりません。そこで、テキスト表示属性を利用して、存在はしているが、表示はされないという方法について説明します。

テキスト表示属性とは、テキスト画面に表示される文字の 1 文字 1 文字に対応して、表示されるモード（点滅、反転など）や色を決定するものです。テキスト表示属性を格納する領域は、A2000H から始まっています。文字部と同様に、2 ページ分の領域が確保されています（2 ページ目は A3000H から）。

テキスト表示属性は 1 バイトで構成されます。しかし、それでは文字部とのアドレス上の対比がとりにくいので、実際には 2 バイトの領域がとられています。よって、セグメントベースを切り替えることにより、文字部と属性部を同じオフセットでアクセスできるわけです。このように、属性には 2 バイトの領域が確保されているにもかかわらず、実際には 1 バイトしか使用されていないのですから、残りの 1 バイトが無駄になります。しかし、この無駄になる部分にはメモリが存在せず、とびとびにメモリが配置され無駄を防いでいます。テキスト表示属性の構成を図 3.2 として示します。

■図 3.2 テキスト表示属性



ここで参考にするのはビット 0 のシークレット属性です。ビット 0 を 0 にすればシークレット属性が効果を現し、対応する文字にはいかなるモードや色が設定されても、画面上には何も表示されません。同時に表示色を黒にしてノーマルモード（反転もブリンクもしないモード）にするのもよいでしょう。

## ■プログラムをグラフィックVRAM上に置く

### ○考え方

VRAM はテキスト用のみではありません。グラフィック用の VRAM は最低でも 96KB 存在し、最大で 256KB にもなるのですから、プログラム領域としても、またデータ領域としても放っておく手はありません。しかも、グラフィック VRAM に置くためのプログラムは、テキスト VRAM のそれより作成が簡単です。しかしテキスト VRAM の場合と異なり、表示するためのデータと実際に VRAM に格納されるデータが異ならないので、暗号化を徹底させるなりして、その扱いを慎重にしなければなりません。



### ○実現方法

とにかくグラフィック VRAM 上に転送すればよいのですから、その方法が複雑であればあるほど効果があります。できれば BIOS などのサービスを用いて、間接的に行うのがよいでしょう。

もちろん、不自然なパターンが表示されることを防ぐために、グラフィック画面の表示を停止しなければなりません。

### ○対処方法

いずれにしても表示がかくされてしまうのですから、プログラム解析によって発見するしかありませんが、発見する方法としては、やはり不自然な画面への表示や、グラフィック VRAM への転送でしょう。すでに説明したとおり、テキスト画面へプログラムを表示するという形式によって格納するには、かなり不自然な文字列を、データとして与えなければなりません。したがって、かなり容易に発見できそうです。

## 3.2 プログラムをスタックに置く

### ○考え方

スタックというのはデータを一時的に待避したり、サブルーチン間でのパラメータの授受に用いる領域ですが、ここをプログラムの置き場所にしてしまおうというわけです。スタックにプログラムが置かれると、解析は実際にスタックに積まれる内容がわからないとできませんから、解析を困難にするには効果的です。

### ○実現方法

まずは、スタックにプログラムを置く方法から考え、実行の方法についてはあとに回しましょう。スタックにプログラムを置くには次の2通りの方法が考えられます。

1つ目は、PUSH 命令によってスタックに命令を積んでいく方法です。用意するプログラムはメモリにあってもよいのですが、命令コードを直接 PUSH 命令によってスタックへ積むほうが、混乱させるには効果的です。しかし、手間がかかります。

例を示してみましょう。次のような短いサブルーチンをスタックに置きます（プログラムの左にあるのは対応する命令コード）。

BE 00 00	MOV	SI,0
BF 00 10	MOV	DI,1000H
B9 00 10	MOV	CX,1000H
F4 A4	REP	MOVSB
CB	RET	

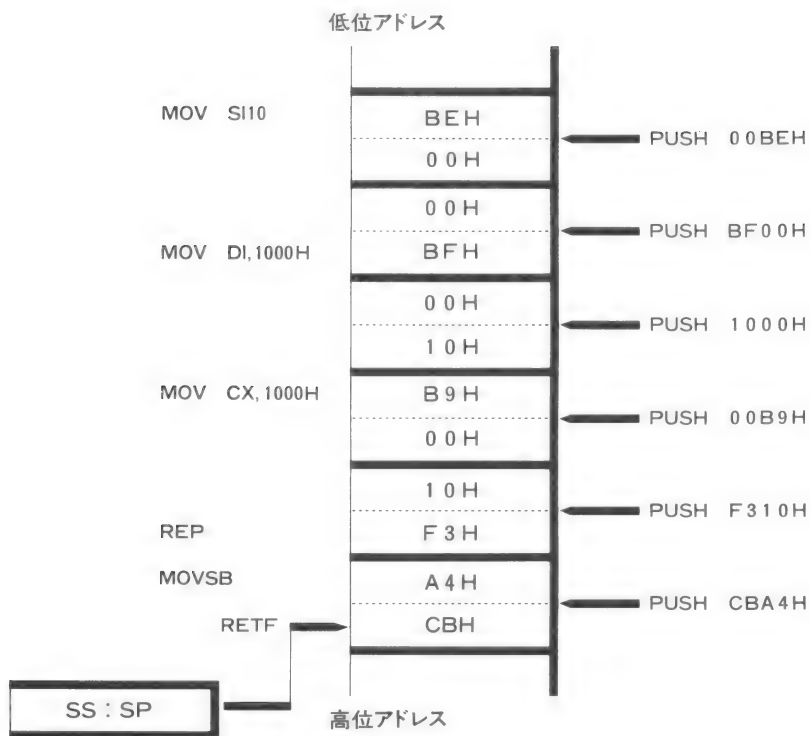
これらのコードを、直接 PUSH 命令を用いてスタックに積むのであれば、そのためのプログラムは以下のようになります。

```
MOV    AX, 0CBA4H
PUSH   AX
MOV    AX, 0F310H
PUSH   AX
MOV    AX, 00B9H
PUSH   AX
MOV    AX, 1000H
PUSH   AX
MOV    AX, 0BF00H
PUSH   AX
MOV    AX, 00BEH
PUSH   AX
```

さて、ここで気を付けなければならないのは、スタックの特性と PUSH 命令の動作です。スタックは 1 個データが積まれるごとに低いアドレスのほうに下がっていきます。要するにスタックにプログラムを積むには、プログラムの末尾のほうから積まなければならないということです。また、PUSH 命令は必ずワードデータをスタックに積みますが、その積まれる順番は上位から下位の順です。すなわち、レジスタ AX がスタックに積まれるとする上記の例ではレジスタ AH、レジスタ AL の順に積まれるわけです。このプログラムの動作を図示すれば、図 3.3 のようになります。

この方法では、スタックに積んでいるデータが定数であるので、それがプログラムとわからなければ、解読は困難になるわけです。

■図 3.3 スタックにプログラムを積む



もっとも、次の例のようにあらかじめメモリ上に存在するプログラムをスタックに積むのであれば、気付くのは早いはずです。

```
LEA BX,PROG-END-1 ;プログラムの末尾-1
MOV CX,6          ;プログラムのサイズ(ワード数)
-LOOP: PUSH [BX]
      SUB BX,2
      LOOP -LOOP
```

ここでレジスタ BX に入れているのは (PROG-END-1)、転送するプログラムの最後の命令より 1 個手前のアドレスです。また、レジスタ CX に入れているのは、命令のバイト数 ÷ 2 の値です (すなわちワード数)。転送を末尾から行っているためにレジスタ BX の内容は減じています。最後の LOOP 命令は、再びスタックヘデータを格納するためのものです。

2 つ目はストリング命令によって、スタック領域に一気にプログラムを転送してしまう方法です。プログラムを転送しているのがはっきりと見えてしまいますので、見破られやすくあまり勧められません。しかし、暗号化と組み合わせればそれなりの効果は得られます。

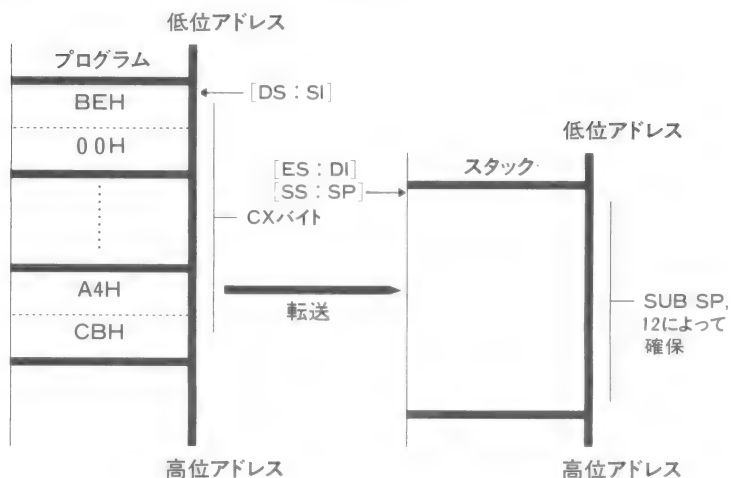
同様に例を示しましょう。プログラムは、転送するターゲットの同じものを用います。

```
SUB    SP,12          ; プログラム領域を確保
MOV     DI,SP
MOV     SI,PROG - TOP
MOV     AX,SS
MOV     ES,AX
MOV     AX,12
REP     MOVSB
```

ここで気を付けることは、最初にレジスタ SP の値を減じていることです。たんに転送を行っても、PUSH 命令のように自動的にレジスタ SP の内容を更新してくれないため、あらかじめレジスタ SP の内容を、プログラムサイズのぶんだけ減じておくのです。これでスタック上にプログラムのための領域が確保されます。また、レジスタ SS の内容をレジスタ ES へコピーしていますが、これは MOVSB 命令が、レジスタ DS, ES をセグメントベースとして転送

を行うのに対し、スタックは、レジスタ SS をセグメントベースとしているからです（図 3.4 参照）。

■図 3.4 スタックにプログラムを転送する



両者共プログラムの実行が終了してしまえば、スタック上の領域は不要になりますから、POP 命令を 6 回行うか、

```
ADD SP,12
```

としてスタックを元に戻します。

さて、プログラムをスタックに置くことができたなら、それを実行しなければなりません。しかし、プログラムがレジスタ CS をセグメントベースとした位置にないため、その実行には工夫が必要です。具体的には、ジャンプ先のアドレス（セグメント：オフセット）をメモリ上に格納しておき、そこに対して、セグメント外 CALL 命令を実行します。手順としては以下ようになります。

- ① メモリ上に、セグメントとオフセットを格納するための領域を確保する
- ② スタック上のプログラムの位置を得て、メモリ上に格納する
- ③ 格納した位置に対してセグメント CALL を実行する

順番に説明しましょう。

①については、メモリ上に他のデータといっしょに確保すればよいでしょう。大きさは4バイトです。

②は、プログラムをスタック上に作成した直後の、レジスタ SS とレジスタ SP の内容をメモリ上に格納します。順番は SP:SS とします。③を含めてプログラム例としてまとめれば、次のようになります。

```

ADDR    DD    ?

        MOV     WORD PTR ADDR,SP
        MOV     WORD PTR ADDR+2,SS
        CALL    ADDR

```

ここでは、呼び出しにセグメント外 CALL を用いているため、スタック上のプログラムは、セグメント外 RET によって復帰しなければなりません。

## 3.3 メッセージをプログラムに

### ○考え方

プログラムを解読しようとして、その先頭に、いきなり意味ありげにメッセージが存在したらどうするでしょうか。

このプログラムはおかしいのではないか？ それともロード方法が違っているのだろうか？ と感じれば、すぐに調べてみましょう。メッセージがプログラムになる場合だってあるのです。

### ○実現方法

基本的には、プログラムの先頭をメッセージにするのです。たとえば以下のようにです。

```
ORG    0

START:

      DB    'Welcome...'
      .....

```

実行開始番地から、いきなりメッセージが置かれていれば、解読していて困惑するでしょう。しかし、なまじメッセージとしてのかたちに捕われているからで、これをプログラムとみなして正直に逆アセンブルし、実行を追ってみれば、きちんとしたプログラムへ続いているかも知れません。そこで、メッセージをプログラムとするための方法について説明しましょう。

メッセージをプログラムとするには、以下の方法が考えられます。



- ① 何もせずに通過する命令を集め、巧妙に組み合わせる
- ② 先頭にジャンプ命令を置く
- ③ プログラムとメッセージが一体化している極めて巧妙なもの

①はメッセージを、特に効果を持たない命令で構成し通過させてしまうものです。主に、メッセージに使用する文字が ASCII キャラクタのうちの英大文字ばかりであれば、特に気にすることもなく、メッセージを書くことができます。それは英大文字のコード 41H～5AH が、INC, DEC, PUSH, POP などの 1 バイトで構成される命令ばかりだからです。しかし、数字が混じれば、そのコード 30H～39H は、XOR, CMP などの命令を表しますが、複数バイトで構成される命令であるため、メモリの一部分を変化させてしまったりする可能性があります。また英小文字が混じれば、そのコード 61H～7AH は未使用命令であるか、条件分岐命令であったりしますので、扱いには注意が必要です。ちなみに 8086 で未使用であっても V30 では有効な場合もあります。さらに注意してください。この条件分岐命令は②と③に関係します。

②は、先頭に故意にジャンプ命令が置かれており、メッセージをスキップしてしまうものです。メッセージが単なる威嚇の意味しか持たないため、あまり効果的でない場合もあります。ここで問題となるのはジャンプ命令のコードで、無条件ジャンプである場合には、ExH が先頭にくるため、ここに対応する文字は、ASCII でグラフィック文字、シフト JIS で第二水準の漢字（さんずい）となっています。共に使用頻度が少ないものですから、自然さを装うにはあまりふさわしくありません。そこで、先ほどの英小文字を用います。フラグの内容が一定していれば、条件分岐命令を用いてジャンプを行うことができます。

③は、ジャンプ命令などによる飛び越しを行わずに、メッセージとプログラムが一致しているという高度なものです。プログラムを構成しつつ、メッセージとしての意味も持たなければなりませんので非常に作成が困難です。

## ○サンプル

①と②について、それぞれ例を示します。

### <例 1>

メッセージ”WELCOME TO THIS WORLD.”がプログラムとして何も意味を持たず、続く位置に存在するプログラムに、そのまま流れが移ることを確かめます。まずメッセージがどのようなプログラムとなるかは、実際に打ち込んでみて、逆アセンブルするのが早いでしょう。

SYMDEB を使用してメッセージを打ち込み、逆アセンブルした例を図 3.5 として示します。

■図 3.5 メッセージを意味のない命令で構成する

```
A>symdeb(a)
Microsoft Symbolic Debug Utility
Version 3.01
(C) Copyright Microsoft Corp 1984, 1985
Processor is [8086]

-e 100 "WELCOME" 60 "TO" 60 "THIS" 60 "WORLD." (a) ----- メッセージを入力。スペースを60H
-d100 (a) ----- ダンプしてみる
3089:0100 57 45 4C 43 4F 4D 45 60-54 4F 60 54 48 49 53 60 WELCOME TO THIS
3089:0110 57 4F 52 4C 44 2E 89 46-00 74 07 3D 81 00 75 14 WORLD.JF.t.=.u.
3089:0120 EB 1D 8A 46 06 30 E4 8B-F0 8A 84 7E 10 30 E4 25 k..F.Od.p..%
3089:0130 04 00 75 08 E8 F3 F8 E8-2D 8C FF 46 04 EB AE 8B .u.hsxh...F.k..
3089:0140 36 AC 0D 8A 04 88 46 06-30 E4 8B F8 8A 85 7E 10 6...F.Od.x...
3089:0150 30 E4 25 02 00 74 08 81-3C 81 40 74 14 EB 1D 8A Od%...t.<.@t.k..
3089:0160 46 06 30 E4 3D 20 00 89-46 00 74 05 3D 09 00 75 F.Od=...F.t.=.u
3089:0170 0B E8 B6 F8 E8 F0 8B FF-46 04 EB C3 83 7E 04 00 .h6xhp...F.kC...

-u100 (a) ----- どのような命令で構成されているか
3089:0100 57 PUSH DI
3089:0101 45 INC BP
3089:0102 4C DEC SP
3089:0103 43 INC BX
3089:0104 4F DEC DI
3089:0105 4D DEC BP
3089:0106 45 INC BP
```

30B9:0107 60	PUSHA		
-u [a]			
30B9:0108 54	PUSH	SP	
30B9:0109 4F	DEC	DI	
30B9:010A 60	PUSHA		—— 特に支障のない命令ばかりだ。 ただしスタックには注意
30B9:010B 54	PUSH	SP	
30B9:010C 48	DEC	AX	
30B9:010D 49	DEC	CX	
30B9:010E 53	PUSH	BX	
30B9:010F 60	PUSHA		
-u [a]			
30B9:0110 57	PUSH	DI	
30B9:0111 4F	DEC	DI	
30B9:0112 52	PUSH	DX	
30B9:0113 4C	DEC	SP	
30B9:0114 44	INC	SP	
30B9:0115 2E894600	MOV	CS:[BP+00],AX	—— ヒリオドはオーバーライド
30B9:0119 7407	JZ	0122	プリフィクスとなる
30B9:011B 308100	CMP	AX,0081	
-q [a]			

さて、先頭から PUSH, INC, DEC 命令が続き、最後に ADD 命令があります。ここで見てもらいたいのはコード 60H で、これは、スペース 20H の代りに用いられているのです。表示自体は変化しませんので、スペースの代用として十分使えるわけです。なぜ 20H を使わないかといえば、20H は AND 命令の 1 バイト目であり、多くの場合、メモリの内容を書き換えてしまい危険だからです。

また最後の 2EH ですが、これはピリオドが対応します。2EH はレジスタ CS によるセグメントオーバーライドプリフィクスですから、続く命令に何が来ても差し支えないわけです。

なお、スタックの値が大幅に変化しますので、いずれレジスタ SP の内容をきちんと設定してやる必要があります。

## <例 2>

メッセージ "welcome to this world!!" の先頭がジャンプ命令になっていることを確かめます。SYMDEB を使用してメッセージを打ち込み、逆アセンブルした例を図 3.6 として示します。

■図 3.6 メッセージの先頭にジャンプ命令を置く

```

A>symdeb[0]
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086]
-e 100 "welcome to this world!!" [0] ----- メッセージを入力、小文字でなければならない
-u 100
30B9:0100 7765          JA      0167 ----- ジャンプ命令だ、実行は後方に移る
30B9:0102 6C          INSB
30B9:0103 63          DB      63
30B9:0104 6F          OUTSW
30B9:0105 6D          INSW
30B9:0106 65          DB      65
30B9:0107 20746F      AND      [SI+6F],DH
30B9:010A 207468      AND      [SI+68],DH
-

```

先頭に JA 命令があり、0167H 番地へのジャンプが行われています。ここで問題となるのは、果して条件が成立するかどうかということですが、この命令の条件を参照すると、

CF OR ZF = 0

となっています。つまり、CF と ZF が双方 0 であるときにのみ、分岐が行われるわけです。幸いなことにプログラム実行開始時には、すべてのフラグがクリアされていますから、条件が成立し分岐が行われることになります。



# 4

## 錯乱のためのテクニック

プログラムの解説を行っている最中に、プログラムがどこかへ行ってしまったり、また、いつのまにかプログラムが入れ替わってしまふというテクニックです。これは暗号化と並んで、すなおな解説が裏目に出るというものです。

## 4.1 自分自身を転送する

### ○考え方

スタック上にプログラムを転送するのに似ていますが、現在実行中である自らのプログラムをどこか遠くへ転送し、そこからまた、実行を始めてしまおうとするものです。解析する立場から見ると、あたかも別のプログラム領域に対して、ジャンプしているかのように見えます。

### ○実現方法

手段としては、単なるブロック転送で十分ですが、実行のつじつまを合わせるためには、ジャンプ方法に少し工夫が必要です。瞬時にジャンプを行うために、隠し命令であるレジスタ CS への、ダイレクト転送命令を用いるのがよいでしょう。

## ○サンプル

プログラムを転送し、MOV CS, AX 命令によってジャンプを行うサンプルを、SYMDEB のオペレーションにより図 4.1 として示します。

■図 4.1 プログラムを転送する

```

A>SYMDEB
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086]

-A
----- 転送するプログラムを入力
2F71:0100 MOV SI,0
2F71:0103 MOV DI,1000
2F71:0106 MOV CX,1000
2F71:0109 CLD
2F71:010A REP
2F71:010B MOVSB
2F71:010C JMP 3071:500 ----- ジャンプ先(現在のCSに100Hを加えたセグメント)
2F71:0111

-A500
----- 転送されるプログラムを入力
2F71:0500 MOV DX,510
2F71:0503 MOV AH,9
2F71:0505 INT 21
2F71:0507 INT 3
2F71:0508

-E510 "PROGRAM WAS TRANSFERED!!" OD OA "$"
----- メッセージを入力
-U3071:500
----- ジャンプ先を逆アセンブル
3071:0500 0000 ADD [BX+SI],AL
3071:0502 0000 ADD [BX+SI],AL
3071:0504 0000 ADD [BX+SI],AL
3071:0506 0000 ADD [BX+SI],AL
3071:0508 0000 ADD [BX+SI],AL
3071:050A 0000 ADD [BX+SI],AL
3071:050C 0000 ADD [BX+SI],AL
3071:050E 0000 ADD [BX+SI],AL
----- 何もない

-G=100
----- 実行
PROGRAM WAS TRANSFERED!!
AX=0924 BX=0000 CX=0000 DX=0510 SP=CF7E BP=0000 SI=1000 DI=2000
DS=2F71 ES=2F71 SS=2F71 CS=3071 IP=0507 NV UP EI PL NZ NA PO NC
3071:0507 CC INT 3
-U3071:500
----- プログラムの所在を確かめる
3071:0500 BA1005 MOV DX,0510
3071:0503 B409 MOV AH,09
3071:0505 CD21 INT 21
3071:0507 CC INT 3
3071:0508 0000 ADD [BX+SI],AL
3071:050A 83C402 ADD SP,+02
3071:050D 5D POP BP
3071:050E C3 RET
-

```

## 4.2 自分自身にかぶせる

### ○考え方

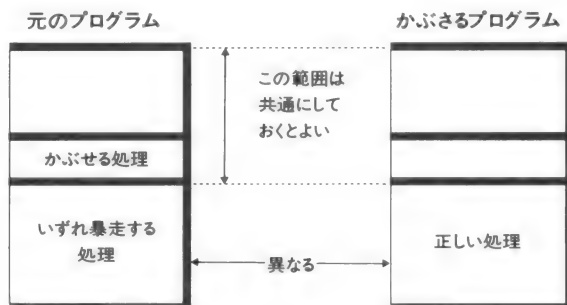
実行中に異なるプログラムを自らの上にかぶせ、かぶさったプログラムを、何ごともなかったかのように実行するものです。

ここでは、かぶせ方がポイントです。

### ○実現方法

ひとえにかぶせ方につきますが、実行をいかに巧みに移すかも重要です。要するに、プログラムが入れ替わってもレジスタ CS、レジスタ IP は変化しませんから、つじつまの合うようにプログラムを配置すればよいのです。ここで、トラップを置くために元のプログラムの終端は無限ループに陥るか、またはどこかへ飛んでいってしまい、暴走してしまうのもよいでしょう。また、かぶさるほうも前半は必要ないわけですが、見破られないために、元のプログラムと同じ内容にしておくのもよいでしょう（図 4.2 参照）。

■図 4.2 プログラムをかぶせる





プログラムのかぶせ方には、いろいろな方法が考えられます。

しかし、たんなるブロック転送ではなくディスクなど、外部から読み込むのがよいでしょう。

### ○サンプル

元のプログラム OVERA.COM が、実行中にかぶさるプログラム OVERB.COM を読み込んで、そこに実行を移す例を、2本のプログラムリストと共に図 4.3 として示します。ここでプログラム OVERA.COM は、ふつうのコマンドとして実行し、プログラム OVERB.COM は不可視属性などを施して、見えないようにしておくといでしょう。

■図 4.3 OVERA. ASM ソースリスト

```

;
;*****
;
;      OVERA.ASM
;
;      プログラムをかぶせるサンプルの、母体となるプログラム
;
;      このプログラムを実行中に、OVERB.COMを自らの上へ読み込み、
;      そちらに実行を移します。
;      カレントドライブ、カレントディレクトリにOVERB.COMのない
;      場合には、プログラムが無限ループに陥りますので注意して下さい。
;
;      COPYRIGHT(C) 1987 BY SHUWA SYSTEM TRADING CO.,LTD.
;
;      LAST MODIFIED ON FEBRUARY 5TH,1987
;
;*****
CODE    SEGMENT
        ASSUME    CS:CODE,DS:CODE,ES:CODE,SS:CODE
;
;      ORG        100H
;
OVERA    PROC
;
;      LEA        DX,OPENING          ; 開始メッセージの表示
;      MOV        AH,9
;      INT        21H
;
;      CALL       READ_PROG          ; プログラムを読み込み、被せる
;
;      INFINITE:
;      MOV        AH,17H              ; ブザーを鳴らす
;      INT        18H
;      XOR        CX,CX

```

```

        LOOP    $
        MOV     AH,18H      ; ブザーを消す
        INT     18H
        XOR     CX,CX
        LOOP    $
        JMP     INFINITE
;
        ORG     200H        ; プログラムの頭を決定するための手段
;
READ_PROG PROC NEAR        ; プログラムを読み込み、被せる
        MOV     AH,3DH      ; ファイルのオープン
        MOV     AL,0
        LEA     DX,OVERB.COM ; 被せるプログラムのファイル名
        INT     21H
        JC      READ_PROG_EXIT ; オープンできないならば終了
;
        MOV     BX,AX        ; ファイルハンドル
        MOV     AH,3FH      ; プログラムの読み込み
        MOV     CX,200H     ; プログラムのサイズ
        MOV     DX,100H     ; 読み込む位置
        INT     21H
        JC      READ_PROG_EXIT ; 読み込めないならば終了
;
        MOV     AH,3EH      ; ファイルをクローズ
        INT     21H
;
READ_PROG_EXIT:
        RET
READ_PROG ENDP
;
OPENING DB 13,10
        DB 'このプログラム実行時にはOVERB.COMが必要です。'
        DB 13,10
        DB 'そのプログラムがない場合、本プログラムは暴走します。'
        DB 13,10,'$'
;
OVERB.COM DB 'OVERB.COM',0 ; 被せるプログラムの名前
;
OVERA ENDP
;
CODE ENDS
;
        END     OVERA

```

#### ■図 4.3 OVERA.COM ダンプリスト

```

00000000 : 8D 16 1F 02 B4 09 CD 21 E8 F5 J0 B4 17 CD 18 33 : 62F
00000010 : C9 E2 FE B4 18 CD 18 33 C9 E2 FE EB EE 00 00 00 : 90F

0020Hから00FFHはすべて00H

00000100 : B4 3D B0 00 8D 16 87 02 CD 21 72 12 8B D8 B4 3F : 695
00000110 : B9 00 02 BA 00 01 CD 21 72 04 B4 3E CD 21 C3 0D : 58A
00000120 : 0A 82 B1 82 CC 83 76 83 8D 83 4F 83 89 83 80 8E : 803
00000130 : C0 8D 73 8E 9E 82 C9 82 CD 4F 56 45 52 42 2E 43 : 775
00000140 : 4F 4D 82 AA 95 48 97 76 82 C5 82 87 81 44 0D 0A : 711
00000150 : 82 BB 82 CC 83 76 83 8D 83 4F 83 89 83 80 82 AA : 8A1
00000160 : 82 C8 82 A2 8F EA 8D 87 81 43 96 78 83 76 83 8D : 8D9
00000170 : 83 4F 83 89 83 80 82 CD 96 5C 91 96 82 B5 82 DC : 8DE
00000180 : 82 B7 81 44 0D 0A 24 4F 56 45 52 42 2E 43 4F 4D : 4C4

```

## ■図 4.3 OVERB. ASM ソースリスト

```

:
: *****
:
: OVERB.ASM
:
: プログラムをかぶせるサンプルの、被さる方のプログラム
:
: このプログラムは、OVERA.COMを実行させる際には、絶対に必要です。
: カレントドライブ、カレントディレクトリにOVERB.COMを置いて
: 下さい。
:
: COPYRIGHT(C) 1987 BY SHUWA SYSTEM TRADING CO.,LTD.
:
: LAST MODIFIED ON FEBRUARY 5TH,1987
:
: *****
CODE    SEGMENT
:        ASSUME    CS:CODE,DS:CODE,ES:CODE,SS:CODE
:
:        ORG        100H
:
OVERB    PROC
:
:        LEA        DX,OPENING          ; 開始メッセージの表示
:        MOV        AH,9
:        INT        21H
:
:        CALL       READ_PROG          ; プログラムを読み込み、被せる
:
:        LEA        DX,ENDING          ; 終了メッセージの表示
:        MOV        AH,9
:        INT        21H
:
:        MOV        AX,4C00H          ; プログラムを無事終了させる
:        INT        21H
:
:        ORG        200H              ; プログラムの頭を決定するための手段
:
READ_PROG PROC    NEAR              ; プログラムを読み込み、被せる
:        MOV        AH,3DH            ; ファイルのオープン
:        MOV        AL,0
:        LEA        DX,OVERB.COM      ; 被せるプログラムのファイル名
:        INT        21H
:        JC         READ_PROG_EXIT    ; オープンできないならば終了
:
:        MOV        BX,AX             ; ファイルハンドル
:        MOV        AH,3FH            ; プログラムの読み込み
:        MOV        CX,100H           ; プログラムのサイズ
:        MOV        DX,100H           ; 読み込む位置
:        INT        21H
:        JC         READ_PROG_EXIT    ; 読み込めないならば終了
:
:        MOV        AH,3EH            ; ファイルをクローズ
:        INT        21H
:
READ_PROG_EXIT:
:        RET
:
READ_PROG    ENDP
:
OPENING DB    13,10
:        DB    'このプログラム実行時にはOVERB.COMが必要です。'
:        DB    13,10
:        DB    'そのプログラムがない場合、本プログラムは暴走します。'
:        DB    13,10,'$'
:
:

```

```

OVERB_COM      DB      'OVERB.COM',0          ; 被せるプログラムの名前
;
ENDING DB      13,10
DB      'プログラムは無事終了しました。
DB      13,10,'$'
;
OVERB  ENDP
;
CODE  ENDS
;
      END      OVERB

```

### ■図 4.3 OVERB.COM ダンプリスト

```

00000000 : 8D 16 1F 02 84 09 CD 21 E8 F5 00 8D 16 91 02 B4 : 636
00000010 : 09 CD 21 88 00 4C CD 21 00 00 00 00 00 00 00 : 2E9

      0020Hから00FFHはすべて00H

00000100 : B4 3D B0 00 8D 16 87 02 CD 21 72 12 8B D8 B4 3F : 695
00000110 : B9 00 01 BA 00 01 CD 21 72 04 B4 3E CD 21 C3 0D : 589
00000120 : 0A 82 B1 82 CC 83 76 83 8D 83 4F 83 89 83 80 8E : 803
00000130 : C0 8D 73 8E 9E 82 C9 82 CD 4F 56 45 52 42 2E 43 : 775
00000140 : 4F 4D 82 AA 95 4B 97 76 82 C5 82 B7 81 44 0D 0A : 711
00000150 : 82 8B 82 CC 83 76 83 8D 83 4F 83 89 83 80 82 AA : 8A1
00000160 : 82 C8 82 A2 8F EA 8D 87 81 43 96 7B 83 76 83 8D : 8D9
00000170 : 83 4F 83 89 83 80 82 CD 96 5C 91 96 82 B5 82 DC : 8DE
00000180 : 82 B7 81 44 0D 0A 24 4F 56 45 52 42 2E 43 4F 4D : 4C4
00000190 : 00 0D 0A 83 76 83 8D 83 4F 83 89 83 80 82 CD 96 : 6E6
000001A0 : B3 8E 96 8F 49 97 B9 82 B5 82 DC 82 B5 82 BD 81 : 98B
000001B0 : 44 0D 0A 24 : 07F

```

### ■図 4.3 OVERA.COM 実行例

A>OVERA [a] ----- OVERB.COM がある状態で実行

このプログラム実行時にはOVERB.COMが必要です。  
そのプログラムがない場合、本プログラムは暴走します。

プログラムは無事終了しました。-----うまくいく

A>DEL OVERB.COM [a] ----- OVERB.COM を削除

A>OVERA [a] ----- 再び実行

このプログラム実行時にはOVERB.COMが必要です。  
そのプログラムがない場合、本プログラムは暴走します。-----暴走した

## 4.3 自分自身を書き換える

### ○考え方

ブロック転送や上書きなどを用いずに、自分自身を小刻みに書き換えていこうというものです。

### ○実現方法

要するに、メモリの一部を書き換えればよいのですが、その書き換え方が問題です。以下のように書き換えてはいけません。

```
MOV     AX,2500H
MOV     BYTE PTR CS:[BX],25H
```

最小限、アドレッシングモードに工夫をこらし、次のようなものにしてしまおう（ここでは、レジスタ CS=レジスタ ES とする）。目的のアドレスや書き込む命令も、定まりにくくなります。

```

CALL    NEXT
NEXT:   PUSH    SP
        XCHG    BP,DI
        MOV     DI,[BP+2]
        LEA     DI,[BP+DI]
        MOV     AL,55H
        MOV     CX,1000H
        REPNE   SCASB
        ADD     CL,0DOH
        MOV     [DI],CL
```



# 5

## ワナをかける

これまでは、プログラムを読みにくくする、つまり、いかにプログラムの解説をさせないかに、焦点を絞って解説してきましたが、ここではプログラムの解説はさせるが、大きなわなが待っているというテクニックについて紹介します。非常にトリッキーなものですので、皆さんは、読んでいてなるほどと思われることでしょう。

## 5.1 書き換えを無効にする

### ○考え方

プログラムを書き換えるというテクニックは、よく用いられるものです。暗号化や自分自身の上にプログラムをかぶせるというのはその一種ですが、これは、その書き換えを無効にするためのテクニックです。リストどおりに解釈したり実行を追ってみても、実際にCPUが行う命令はそれとは異なるのです。

### 実現方法

8086にはプリフェッチキューというものがあります（資料編を参照）。ここでプリフェッチキューの働きについて詳しく説明すると、以下ようになります。

8086は、大きく2つのブロックに分けることができます。1つ目は実行ユニット（EU）、2つ目はバスインタフェースユニット

(BIU) です。簡単にいえば、EU は命令を実行するブロック、BIU は命令の取り出しを含むメモリアクセスを行うブロックです。

ここで注目するのは EU と BIU の連携動作で、図 5.1 (8086 における 3 個の命令の実行のようす) にも示すように、EU が働いている間は BIU はひまなのです。特に除算命令などの長大な実行時間を要するものは、この傾向が顕著に現れます。

この、ひまな時間を利用して、あらかじめ次に実行するであろう命令コードを取り込んでおくことをプリフェッチといい、取り込んだ命令コードを格納する場所をプリフェッチキューといいます。

■図 5.1 EU と BIU の動作

命令	ステップ	EU	BIU	バス
MOV [BX], AL	実行		フェッチ 	アクティブ 
	書き込み		書き込み 	アクティブ 
DIV [BX]	読み出し		読み出し 	アクティブ 
	実行		フェッチ 	アクティブ 
ADD [DI], AL	読み出し		読み出し 	アクティブ 
	実行		フェッチ 	アクティブ 
	書き込み		書き込み 	アクティブ 

図 5.1 について説明すると、まず除数をレジスタ BX によって示されるアドレスからロードします。このとき BIU はアクティブになり、メモリ上から要求されるデータを取り出して EU に転送します。EU はそれを受けて、内部レジスタを用いて除算を実行しますが、それにはけっこう時間がかかります。この間に BIU はプリ

フェッチキューの状態を見て、空きがあれば命令コードを前もってプリフェッチキューに取り込んでおくのです。このようにプリフェッチを行っておけば、除算命令が終了し、次の命令を実行する段階において、あらためて、BIU にメモリアクセスを行わせる必要がなくなります。メモリアクセスは CPU 全体の処理から見た場合、比較的時間のかかる部類に属する仕事ですので、大きなメリットになるわけです。

さて、話を元に戻しましょう。ここでおさえておいてほしいのは、プリフェッチキューに蓄えられている命令コードは、JMP 命令などのプログラムの流れを変える命令に出会わない限り、必ず実行されるという点です。これを利用すれば、たとえ、実行中の命令の次の命令を書き換えても、プリフェッチキューには、すでに書き換える前の命令が入っていますから、そちらが優先されて実行されるわけです。

具体的に確かめてみましょう。まず SYMDEB の A,E コマンドを用いて、図 5.2 のようなプログラムを作成してみました。これは、ある位置で直後の命令を書き換え、書き換わった結果が有効であれば、“Program terminate normally(1)”とメッセージを表示して (SYMDEB が表示)、また書き換わった結果が無効であれば、“TRAPPED.”のメッセージを表示してそれぞれ正常終了するものです。図 5.2 中のプログラム、および実行手順を追ってください。

結果は、まず後者になるのがふつうです。この例において、プログラムの先頭で割り込みを禁止している点に注意してください。命令の実行中に割り込みが入ると、そこで割り込み処理ルーチンへのジャンプが行われるわけですから、プリフェッチキューの内容が破棄されてしまいます。



## ■図 5.2 書き換えを無効にする

```

A>SYMDEB
Microsoft Symbolic Debug Utility
Version 3.01
(C) Copyright Microsoft Corp 1984, 1985
Processor is [8086]
-A ————— テスト用のプログラムを作成
30B9:0100 MOV BX,106
30B9:0103 MOV BYTE [BX],C3 ————— 次のアドレスへ"C3H"を書く
30B9:0106 MOV AH,9
30B9:0108 MOV DX,200
30B9:010B INT 21
30B9:010D INT 3
30B9:010E
-E200 "TRAPPED!!" OD 0A "$" ————— メッセージを入力
-G=100 ————— ふつうに考えればそのまま終了するが...
TRAPPED!! ————— プログラムは継続した
AX=0924 BX=0106 CX=0000 DX=0200 SP=CE36 BP=0000 SI=0000 DI=0000
DS=30B9 ES=30B9 SS=30B9 CS=30B9 IP=010D NV UP EI PL NZ NA PO NC
30B9:010D CC INT 3
-U100 ————— 確認
30B9:0100 B80601 MOV BX,0106
30B9:0103 C607C3 MOV Byte Ptr [BX],C3
30B9:0106 C3 RET ————— 確かに書き換わっている
30B9:0107 09BAA002 OR [BP+SI+0200],DI
30B9:010B CD21 INT 21
30B9:010D CC INT 3
30B9:010E 7E10 JLE 0120
30B9:0110 30FF XOR BH,BH
-

```

## ○対処方法

現在、実行中のアドレスの直後を書き換えているようであれば、プリフェッチキューを利用してプロテクトがかけられている可能性があります。ここで問題となるのは、何バイト先まで有効なのかということです。8086のプリフェッチキューは6バイトですから、最長でも6バイト先まで見ればよいことになります。そうすると、ほとんど1命令ないしは2命令とみなせます。

もっとも、このようなプロテクトをかけられた場合、かえて書き換えた結果を考えなくてもよいということになるため、この方法は気付きさえすればかえて単純だといえましょう。

## 5.2 バンク切り替えを使う

### ○考え方

PC-9801 には、グラフィック VRAM が 2 ページ実装されており（ただし PC-9801/U を除く）、これらはバンク切り替えによってどちらかを選択してアクセスするようになっています。当然ここにもプログラムを置くことができますが、ただ置いたのではおもしろくありません。そこで、2 ページあるグラフィック VRAM のそれぞれにプログラムを置き、それらを切り替えて使おうというわけです。

### ○実現方法

PC-9801 では、グラフィック VRAM のバンクを I/O アドレス A6H で切り替えます。グラフィック VRAM のバンク番号を 0, 1 とすれば、A6H に 0 を出力したときはバンク 0 が、A6H に 1 を出力したときにはバンク 1 が、それぞれ選択されます。具体的なオペレーションとしては、

```
MOV     AL,<バンク番号>
OUT     OA6H,AL
```

となります。A6H のほかに A4H もありますが、これは CPU からのアクセスには特に関係なく、表示されるバンクを切り替えるためのもの、すなわち GDC に対するためのもののなのです。

グラフィック VRAM には、それぞれ独立したプログラムを置き、それを、メイン RAM 上から呼び出して使用しても十分効果があります。それはプログラムを解読していても、プログラムで選択さ

れるべきバンクと、実際に解釈されるバンクが一致しない可能性があるからです。しかし最も効果的なのは、プログラムをグラフィック VRAM 上で動作させている間に、グラフィック VRAM のバンクを切り替えてしまうものです。これだと、仮にバンク 0 でプログラムが走っている最中でも、バンク 1 へ切り替えられてしまえば、実行はバンク 1 上のプログラムへ移ります。このとき、バンク 0 のプログラムとバンク 1 のプログラムが互いに連携がとられていれば、何事もなかったように実行が継続されるわけです。

しかし、5.1 でも触れたように、8086 にはプリフェッチキューというものが存在します。その意味するところは、たとえバンクを切り替えてもプリフェッチキューには命令が残っており、しばらくは切り替える前のバンクのプログラムが実行されるということです。

ここで問題となるのは、いったい何バイトの命令が残るのかということですが、これについては図 5.3 のような実験をしてみました。参考にしてください。

まず実験用の環境を作成しますが、これは SYMDEB を用いてグラフィック VRAM 上に直接作成しています。その前に、環境を整えるため PSP をバンク 0・バンク 1 の両方にコピーし、セグメントレジスタ CS, DS, ES の内容をグラフィック VRAM のセグメントベースである A800H にセットしています（レジスタ SS は変更しないでください。レジスタ SP も同様です）。バンクの切り替えは O コマンドを用いています。

次にプログラムを作成しましたが、これもバンク 0 とバンク 1 の両方に作成しています。

ここで作成したプログラムに説明を加えますと、バンク 0 のプログラムは、レジスタ CX をクリアしたあとにバンク切り替えを行い、“INC CX”を 6 回実行して、バンクが切り替わっても何回レジスタ CX が増加されるかを、調べるためのものです。

■図 5.3 バンク切り替えの実験

```

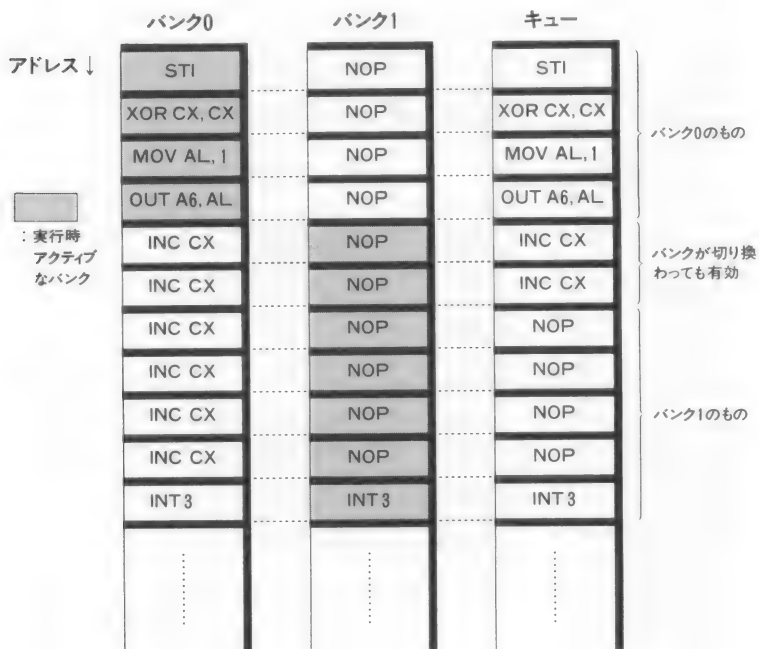
A>SYMDEB
Microsoft Symbolic Debug Utility
Version 3.01
(C) Copyright Microsoft Corp 1984, 1985
Processor is [8086]

-OA6 0                バンク0を選択
-MO FF A800:0         PSPを作成
-OA6 1                バンク1を選択
-MO FF A800:0         PSPを作成
-RCS
CS 30B9
: A800
-RDS
DS 30B9
: A800
-RES
ES 30B9
: A800
-OA6 0                バンク0に戻す
-A100                 バンク0に置くプログラム
A800:0100 STI
A800:0101 XOR CX,CX    カウンタ
A800:0103 MOV AL,1
A800:0105 OUT A6,AL
A800:0107 INC CX
A800:0108 INC CX
A800:0109 INC CX
A800:010A INC CX
A800:010B INC CX
A800:010C INC CX
A800:010D INT 3
A800:010E
-
-OA6 1                バンク1へ
-A100                 バンク1に置くプログラム
A800:0100 NOP
A800:0101 NOP
A800:0102 NOP
A800:0103 NOP
A800:0104 NOP
A800:0105 NOP
A800:0106 NOP
A800:0107 NOP
A800:0108 NOP
A800:0109 NOP
A800:010A NOP
A800:010B NOP
A800:010C NOP
A800:010D NOP
A800:010E NOP
A800:010F NOP
A800:0110 INT 3
A800:0111
-OA6 0                バンク0へ戻す
-G=A800:100           実行 2回実行された /
AX=0001 BX=0000 CX=0002 DX=0000 SP=CE36 BP=0000 SI=0000 DI=0000
DS=A800 ES=A800 SS=30B9 CS=A800 IP=0110 NV UP EI PL NZ NA PE NC
A800:0110 CC          INT 3
-U100
A800:0100 90          NOP
A800:0101 90          NOP
A800:0102 90          NOP
A800:0103 90          NOP
A800:0104 90          NOP
A800:0105 90          NOP
A800:0106 90          NOP
A800:0107 90          NOP

```

バンク 1 のプログラムは、先頭から 10 数バイトは“NOP”で続く  
 “INT3”命令によって実行終了するものです。実行のようすを図示  
 すると、図 5.4 のようになります。

■図 5.4 バンク切り替え実行のようす



実行してみるとプログラムは正常に終了しますが、そのときに、  
 レジスタ CX の内容を確認すると 0002H となっています。すなわ  
 ちバンク切り替えを行っても、続く 2 バイトは有効であることを意  
 味しているのです。同時に、切り替わったほうのバンクの 2 命令は、  
 実行されないことを意味します。ただし、これは V30 で実験した  
 場合であり、8086,80286 では 3 バイトという結果が出ました。これ

は CPU の内部サイクルの差と思われます。一般的にはバンク 0 のプログラム、バンク 1 のプログラムの両方に、“NOP”命令による緩衝帯を設けるとよいでしょう。しかし、これではヒントを与えていることになります。バンク切り替えによる実行の流れと、解説による実行の流れの異なることが望まれます。

具体的には、バンク切り替えの直後に 2 バイトのジャンプ命令を入れるとよいでしょう。

#### ○対処方法

5.1 と同様です。グラフィック VRAM 上で動作しているプログラムがバンクを切り替えるような動作をしたら、その直後の命令に気を付けなければなりません。

## 5.3 タイマ割り込みを使う

#### ○考え方

プログラムリストを追ったり、また実際にプログラムを実行させて追うときは、一般にそのプログラム以外の流れは気にしないものです。この方法はこのような盲点を突いたもので、ふいにプログラム実行の流れを変えてしまうものです。

#### ○実現方法

タイマ割り込みを利用します。つまり、プログラムリスト上の流れ（これをメインストリームと呼びます）が実行されている間に、タイマによる割り込みを発生させて、実行をその割り込み処理ルーチンへ移してしまうのです。この移行は、プログラムリスト上では

察することが困難ですので、目くらましには最適です。

具体的な例を示す前に、タイマ割り込みの発生のおさせ方について説明しておきましょう。PC-9801では、タイマ割り込みを発生させる LSI に  $\mu$ PD8253C が搭載されています。 $\mu$ PD8253C は 3 つのカウンタを持ち、機能は表 5.5 のように割り当てられています。

■表 5.5  $\mu$ PD8253C の 3 つのカウンタの機能

番 号	PC-9801/E/F/M	PC-9801U/VF/VM/UV/VX
カウンタ0	インターバルタイマ	インターバルタイマ
カウンタ1	DRAMリフレッシュ	内蔵スピーカ周波数設定
カウンタ2	RS-232C	RS-232C

ここで使用するのは、カウンタ 0 のインターバルタイマ機能です。 $\mu$ PD8253C に適切なプログラミングを施せば、すぐにインターバルタイマが起動されますが、PC-9801 ではタイマ BIOS が提供されており、これを使用すればさらに起動が容易になります。ここではタイマ BIOS の使用法について説明しましょう。

タイマ BIOS は、割り込み命令“INT 1CH”によって呼び出します。このとき、以下のパラメータを与えます。

レジスタ AH ← 02H

レジスタ CX ← タイムアウト値

レジスタ BX ← タイムアウト時の処理ルーチンのアドレス  
(オフセット)

レジスタ ES ← タイムアウト時の処理ルーチンのアドレス  
(セグメント)

インターバルは 10ms に固定されていて、この時間が経過するたびに“INT 08H”の割り込みが発生します。このとき、レジスタ CX に設定された値だけ割り込みが発生したら、レジスタ ES:BX によって指定されるタイムアウト処理ルーチンへ制御を移すことができます。ここでは、このタイムアウトの検出機能によって制御を移すのです。

具体例を示しましょう。実は、この方法で制御を移すには時間的な計算が必要となります。つまり、タイマを起動した段階で制御が移る時間がわかるわけですが、そのとき、制御が移る前に行っておかなければならないすべての処理は、終了していなければならないのです。必要な処理が終わっていれば、プログラムは無意味な作業を行って、そのうちに割り込みが入るのを待てばよいのです。この無意味な作業には、何か意味ありげな複雑な演算などがよいのですが、ここでは例を簡単にするため、ダミーループを設けておき、ループの途中で割り込みによって脱出することにします。

さて、図 5.6 の SYMDEB による例を参照してください。

プログラムは先頭でタイマを起動させ、あとはレジスタ CX を二重に用いたダミーループを行っています。ダミーループが終了すればプログラムは終了し、SYMDEB のコマンド待ちとなります。

しかし、1 秒 (=1000ms) 後にタイムアウト割り込みが入ることになっていますから、続く領域に用意してあるブザーを 5 回鳴らすルーチンへ制御が移り、そこで無限ループに陥ってしまいます。



■図 5.6 インターバルタイマによる制御の移行

```

A>SYMDEB
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [8086]
~A100
2F79:0100 MOV AH,2
2F79:0102 MOV CX,64
2F79:0105 MOV BX,200
2F79:0108 INT 1C
2F79:010A MOV CX,10
2F79:010D PUSH CX
2F79:010E XOR CX,CX
2F79:0110 LOOP 110
2F79:0112 POP CX
2F79:0113 LOOP 10E
2F79:0115 MOV AX,4C00
2F79:0117 INT 21
2F79:0119
~A200
2F79:0200 MOV CX,5
2F79:0203 PUSH CX
2F79:0204 MOV AH,17
2F79:0206 INT 18
2F79:0208 XOR CX,CX
2F79:020A LOOP 20A
2F79:020C MOV AH,18
2F79:020E INT 18
2F79:0210 XOR CX,CX
2F79:0212 LOOP 212
2F79:0214 POP CX
2F79:0215 LOOP 203
2F79:0217 JMP 217
2F79:0219
~G=100

```

1秒後に割り込みが入るよう設定

ブザーを5回鳴らす

ループ

1秒後にブザーがなる

## ○対処方法

このような方法でプログラムを実行している場合には、次の3点に注意します。

- ① インターバルタイマを起動していないか
- ② 無意味であると思われる動作を繰り返していないか
- ③ タイムアウト時の行方（インターバルタイマを起動している場合）

## 5.4 スタックを書き換える

### ○考え方

“わな”をかけるための最後のテクニックは、スタックを書き換えてしまうというものです。スタックには、サブルーチンからの戻りアドレスや各種レジスタの内容が積まれていることが多く、非常に重要な領域といえます。ここを操作してサブルーチンからの戻り先を変えたり、レジスタの内容を変えてしまおうというものです。

### ○実現方法

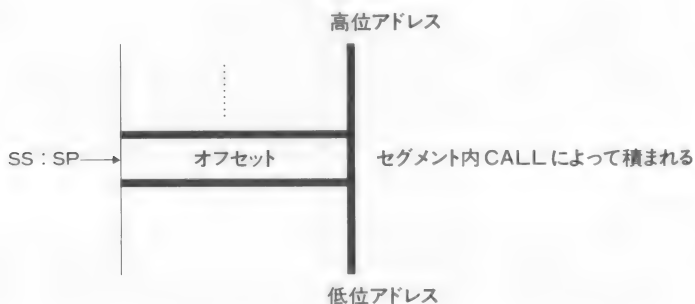
原理的には、ハードウェアを操作するでもなく、BIOS を使用するわけでもなく非常に単純です。まずは、戻りアドレスのほうから変えてみましょう。

今まで説明してきたとおり、サブルーチンの呼び出し（CALL 命令、INT 命令など）においては、戻るべきアドレスがスタックに積まれます。本来ならばここには、CALL 命令か INT 命令を実行した位置の、次の位置があるはずなのですが、ここを別の位置へ書き換えてしまうと、RET 命令、あるいは IRET 命令によって、本来戻るべきところには戻らず、どこか別のところへ戻ってしまいます。そして、この特性を利用し、多少手のこんだジャンプ命令にしてみましょうのです。

原則として、スタックの操作はサブルーチン側で行います（というよりサブルーチンの側でしか操作できません）。サブルーチンが呼ばれた直後には、レジスタ SP の指す位置に戻るべき位置のオフセットが積まれています。ただし、これはセグメント内 CALL の場合で、セグメント外 CALL の場合にはさらにセグメントベース

が、INT 命令の場合には、さらにフラグレジスタが積まれています。ですから自分が呼ばれた形式を認識し、それに合った書き換えを行わなければなりません。ここでは、セグメント内 CALL が行われたものとします。よって、書き換えは 1 ワードです。

■図 5.7 サブルーチンが呼ばれた際のスタック



まずはサブルーチンの定型を示しましょう。特に何も行わないのであれば、おおかた次のようになります。ここでは複雑さを増すために、何やら意味のありそうなことを行ってみましょう。

```
MOV    BP,SP
MOV    AX,<戻りオフセット値>
MOV    [BP],AX
RET
```

これだけで OK です。先頭でレジスタ SP の内容をレジスタ BP にコピーしていますが、これはレジスタ SP がメモリへの代入などに使えないからです。レジスタ BP は、特に指定しない限りレジスタ SS をセグメントベースとしたメモリアクセスを行いますから、スタックを操作するなどという用途に適しているわけです。

次はレジスタの内容を書き換えてみましょう。この場合にはサブルーチン呼び出す側とサブルーチンの側で、スタックにはどのようにレジスタが積まれているのか、共通の認識を持つ必要があります。

サブルーチン呼び出す側では、サブルーチン呼び出しの際にはレジスタを待避したのですが、サブルーチンの側でそれを書き換えてしまったため、結果として、サブルーチンがレジスタを破壊したのと同じ効果を持つのです。すなわち待避が無駄になるわけで、待避が正常なものとして解釈を行うと、どこかでつじつまが合わなくなります。

基本的には戻り位置の書き換えと同じです。ただし、サブルーチン呼び出しより前にスタックに積まれたデータは、戻りアドレスより高位アドレスに存在しますから、スタックへの積み方を考慮して、アドレスを計算してやらなければなりません。仮にサブルーチン呼び出す前に、レジスタ AX, BX, CX, DX の4つのレジスタを順にスタックに積んだとしましょう。このとき、サブルーチンが呼ばれた際のスタックのようすは、図 5.8 のようになります。

■図 5.8 サブルーチンが呼ばれた際のスタック

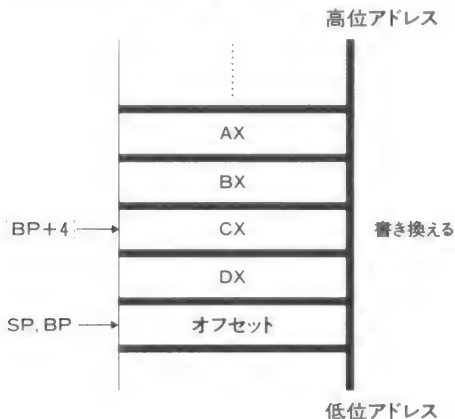


図 5.8 から明らかなですが、戻りアドレスの上にあるのはレジスタ `DX` で、最も遠くにあるのはレジスタ `AX` です。ここでレジスタ `CX` のみを書き換えなければ、レジスタ `SP` の内容に 4 を加えたものを書き換えのアドレスとすればよいわけです。このとき戻りアドレスを書き換えたように同様の例を示せば、次のようになります。

```
MOV    BP,SP
MOV    AX, [BP+4]
ADD    AX,10
MOV    [BP+4],AX
RET
```

レジスタ `AX` を経由して、結果的にレジスタ `CX` の内容が 10 増すようになっています。すなわちサブルーチンの呼び出しの前後で、スタックに待避したはずのレジスタ `CX` の内容が、変化してしまうわけです。

### ○対処法

サブルーチン内部で、不要なスタックアクセスを行っていないかをチェックします。特に、戻りアドレスを越えてのアクセスが書き込みである場合には要注意です。

# 6

## 既存の知識を破棄させる

プログラムを動かしてさまざまな機能を使用するとき、それが標準的なシステムであれば、機能の詳細や使用法は、マニュアルやその他の資料によって明かです。しかしそのことは、他人に理解されないプログラムを書くという点では不利になります。ここでは、すでにあるシステムを標準的なものではないようにし、既存の知識を無駄にするテクニックを紹介します。

## 6.1 割り込みベクタをすり替える

### ○考え方

PC-9801 の持つさまざまな機能の多くは、ソフトウェア割り込み命令 INT によって呼び出されます。INT 命令のタイプに対応した機能というのは、OS 添付のマニュアルや市販の資料類によっても明らかにされていますから、これを、そのままのかたちで用いたのでは不利になります。そこで、標準のシステムから INT 命令のタイプと機能の対応を変化させて、解読をより困難にしようというものです。

### ○実現方法

ここで問われるのは考え方です。特にテクニックが問われるものではありません。既存の割り込みベクタの内容と、別の空いているベクタの内容とを交換すればよいのです。たとえば、有名な割り込

みである“INT 1BH”に対応したベクタの内容を、0F0Hのものへ変更して、代りに0F0Hのベクタの内容を1BHに設定します。次に具体的なプログラムを示しましょう。

```

XOR      AX,AX           ; 割り込みベクタのセグメントを設定
MOV      DS,AX
MOV      SI,2BH * 4      ; 1BHに対応したベクタアドレス
MOV      DI,0F0H * 4     ; 0F0Hに対応したベクタアドレス
MOV      AX, [SI]        ; オフセットアドレスを交換
XCHG     AX, [DI]
MOV      [SI],AX
MOV      AX, [SI+2]       ; セグメントベースを交換
XCHG     AX, [DI+2]
MOV      [SI+2],AX

```

これでベクタの交換が完了します。ベクタの交換を行う部分が多少複雑ですが、直線的なプログラムですので、すぐに理解することができます。

この手続きを行っておけば、今まで“INT 1BH”で呼び出していたディスク BIOS に関連した機能も、“INT 0F0H”で呼び出せるようになります。また、この状態で“INT 1BH”を行えばタイプ0F0Hの機能が呼び出されますが、MS-DOSでは、ほぼ必ず“Int trap halt”のメッセージを表示し、システムは停止します。

### ○対処方法

まず、割り込みベクタが交換、もしくは書き直されていることに気付かなければなりません。次に、割り込みベクタを交換している箇所を捜すことです。多くの場合、書き換えはトリッキーな手段を用いていることが考えられますから、決して容易ではありません。

## 6.2 パラメータインタフェースを変える

### ○考え方

INT 命令によって BIOS などの機能呼び出す場合、必ずといってよいほどパラメータを指定します。多くの場合、パラメータはレジスタに与え、そうでない場合でも、パラメータのある位置をレジスタに示させるというものは多いようです。割り込みのタイプに対応した機能と同様、パラメータもマニュアル類で公開されていますが、この対応を崩せば、何が行われているかを推測することは難しくなります。割り込みタイプのすり替えを併用すればさらに効果的でしょう。

### ○実現方法

単純に書き換えればよいという問題ではなく、多少、処理内容が複雑になりますが、割り込みのロギングと同じ原理で、処理をいったん横取りし、必要な処理を行ったあとに通常の処理へ戻します。パラメータインタフェースの変更は、レジスタの交換で行うだけの簡単なものとします。

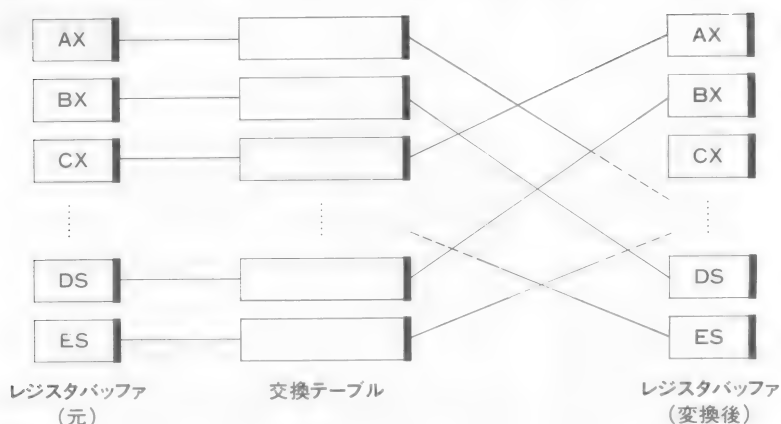
ロギングを行う部分は基礎編で書いていますので、ここではレジスタの交換についての説明を行い、簡単な例を示すことにします。

レジスタ交換を行うための簡単な方法は、交換テーブルを設けるものです。すなわち、呼び出し時のレジスタの内容を交換テーブルに入れ、交換テーブルから正規のレジスタへ内容を転送し、通常の割り込み処理に入ります。交換テーブルは、レジスタの内容を入れておくためのバッファと、対応を示したベクトルで構成されます。

これを図示しますと、図 6.1 のようになります。



■図 6.1 交換テーブル



交換はもっと複雑な過程を経て行ったほうがよいのですが、ここでは説明のため簡略化してあります。

さて、ここでのプログラム例はロギングを行ったとして作成しました。すなわち、本来の割り込みベクタの内容が待避され、新たな割り込みベクタの内容が設定されているとします。プログラム例としては、次ページ図 6.2 のようなものが適当でしょう。

呼び出し時のレジスタ値を格納するテーブルと、変換後のレジスタ値を格納するテーブルを用意し、さらに変換規則を置いたベクタテーブルを用意して交換テーブルを形成しています。

この方法は、割り込みのタイプを変更してから行うとさらに効果的でしょう。

■図6.2 ログイングを行ったプログラム例

```

... ..
REGS    STRUC    ; レジスタパックの構造を定義
_AX     DW       ?
_BX     DW       ?
_CX     DW       ?
_DX     DW       ?
_SI     DW       ?
_DI     DW       ?
_BP     DW       ?
_DS     DW       ?
_ES     DW       ?
REGS    ENDS

... ..
ADDR    DD       ? ; 本来の割り込みベクタの内容が待避されている
... ..

SWAP_DATA1 DW 9 DUP(?) ; 呼び出し時のレジスタ
SWAP_DATA2 DW 9 DUP(?) ; ジャンプ時のレジスタ
SWAP_VECTOR DW SWAP_DATA2+2 ; AX→BX
            DW SWAP_DATA2+10 ; BX→DI
            DW SWAP_DATA2+8 ; CX→SI
            DW SWAP_DATA2+4 ; DX→CX
            DW SWAP_DATA2+16 ; SI→ES
            DW SWAP_DATA2+12 ; DI→BP
            DW SWAP_DATA2+6 ; BP→DX
            DW SWAP_DATA2+14 ; DS→DS
            DW SWAP_DATA2 ; ES→AX

... ..
ENTRY:   ; 割り込みのエントリ（ベクタに設定されている）
MOV      SWAP_DATA1._AX,AX ; 各レジスタのセーブ
MOV      SWAP_DATA1._BX,BX
MOV      SWAP_DATA1._CX,CX
MOV      SWAP_DATA1._DX,DX
MOV      SWAP_DATA1._SI,SI
MOV      SWAP_DATA1._DI,DI
MOV      SWAP_DATA1._BP,BP
MOV      SWAP_DATA1._DS,DS
MOV      SWAP_DATA1._ES,ES

;
LEA      SI,SWAP_DATA1 ; もとのレジスタ値があるテーブル
LEA      BX,SWAP_VECTOR ; 変換用のベクトル
MOV      CX,9 ; レジスタの数

;
_LOOP:   MOV      AX,[SI] ; レジスタ値を得る
        MOV      DI,[BX] ; 格納先のアドレスを得る
        MOV      [DI],AX
        ADD      SI,2
        ADD      BX,2
        LOOP     _LOOP

;
MOV      AX,SWAP_DATA2._AX ; 各レジスタの復帰
MOV      BX,SWAP_DATA2._BX
MOV      CX,SWAP_DATA2._CX
MOV      DX,SWAP_DATA2._DX
MOV      SI,SWAP_DATA2._SI
MOV      DI,SWAP_DATA2._DI
MOV      BP,SWAP_DATA2._BP
MOV      DS,SWAP_DATA2._DS
MOV      ES,SWAP_DATA2._ES

;
JMP      ADDR ; 本来の処理へジャンプ

```





# 7

## プログラム実行のテクニック

プログラムを、ただ実行するのではおもしろくありません。ここでは、プログラムを実行する上で考えられるテクニックについて紹介します。

## 7.1 プログラムを並行実行する

### ○考え方

プログラムというものは多くの場合、頭から終りに流れます。

またジャンプ命令などがあれば、指定されるアドレスに流れるのがふつうです。実行されるプログラムは1個だけで、ふつうは2個以上のプログラムを同時に実行させることはできません。しかし大局的に見れば、同時に2個以上のプログラムを走らせることは可能です。それは実際にマルチタスクというかたちで実現されています。ここでは、プログラムを同時に複数個走らせるテクニックについて紹介します。

### ○実現方法

あまり大規模なものは、目的からいっても、あまり好ましくありません。用途を限定した簡易なものについて実現方法を示します。これはシングルステップ割り込みを用いたものです。

いままで見てきたとおりシングルステップ割り込みとは、プログラムを1命令実行することに発生する割り込みのことで、TFを1として発生させることができます。ここでは、シングルステップ割り込み処理ルーチンをマネージャ（管理を行うプログラム）として、プログラムの並行実行を可能にします。

原理を簡単に説明します。まず、プログラムA、Bと3つのプログラムがあったとします。それぞれが実行開始アドレス、フラグ初期値、レジスタ初期値をメモリ上に保持しておき、実行が開始されるべきアドレスを示しています。ここで、仮にプログラムAが実行されるとします。プログラムAが実行されると、1命令実行が終了したところで、シングルステップ割り込み処理ルーチン（以降Cとします）へ制御が移りますから、ここでは、プログラムAへの戻りアドレスとフラグをメモリ上へ待避します（先ほど実行開始アドレスを設定したところ）。戻りアドレスとフラグは、スタックから取り出すことができます。また、各レジスタの内容もメモリに待避しておきます。そこでプログラムBに対する情報をスタックやレジスタへ戻し、プログラムBへ実行を移します。プログラムBが2命令実行を終了すれば、再び、プログラムAについて同様のことを繰り返します。ここで問題となるのはスタックですが、プログラムA、Bでは、異なる領域をスタックに設定しておかなければなりません。またCでは、あらゆるレジスタの破壊を禁じ、かつ割り込みを禁止しなければなりません。

### ○対処方法

同時に、プログラムが2個以上動いていることを見つけなければなりません。なお、この方法を用いている（手段は違う）ものは、最終的に、シングルステップ割り込み処理ルーチンからどこかへ、ジャンプしてしまいます。

## ○サンプル

同時に3つのプログラムを動作させるサンプル SINGLE.COM を図7.1として示します。実行すると画面の上下から中央に向かって画面が乱れ、中央から元に戻っていきます。つまり、画面の上下を異なるルーチンで制御しているのです。

■図 7.1 SINGLE.ASM ソースリスト

```

:
: *****
:
: SINGLE.ASM
:
: シングルスステップ割り込みを、タスク切替に使用するサンプル
:
: 画面が上下両方から変化し、またもとへ戻っていきます。これを
: 並行実行で実現しています。
: 今回は、処理を単純にするためにスタックに関しては供用と
: している。
:
: COPYRIGHT(C) 1987 BY SHUWA SYSTEM TRADING CO.,LTD.
:
: LAST MODIFIED ON FEBRUARY 5TH,1987
: *****
:
: INFO      STRUC          ; タスクごとの固有情報の格納領域の構造
:_AX        DW             ? ; レジスタAX
:_BX        DW             ? ; レジスタBX
:_CX        DW             ? ; レジスタCX
:_DX        DW             ? ; レジスタDX
:_SI        DW             ? ; レジスタSI
:_DI        DW             ? ; レジスタDI
:_BP        DW             ? ; レジスタBP
:_IP        DW             ? ; レジスタIP
:_CS        DW             ? ; レジスタCS
:_DS        DW             ? ; レジスタDS
:_ES        DW             ? ; レジスタES
:_FLAGS     DW             ? ; フラグレジスタ
:INFO      ENDS
:
: CODE      SEGMENT
:           ASSUME        CS:CODE,DS:CODE,ES:CODE,SS:CODE
:
:           ORG           100H
:
: SINGLE    PROC
:           JMP           MAIN
:
: CUR_TASK  DW             ? ; 現在動作させているタスク
: TASKA_INFO DB           (SIZE INFO) DUP(0) ; タスクAの情報
: TASKB_INFO DB           (SIZE INFO) DUP(0) ; タスクBの情報
:
: MAIN:
:           LEA            DX,OPENING          ; 開始メッセージの表示
:           MOV            AH,9
:           INT            21H
:
:           MOV            AH,25H              ; シングルスステップ割込のベクタを変更する

```

```

MOV     AL,1
LEA     DX,SINGLESTEP
INT     21H

;
LEA     SI,TASKA_INFO
LEA     DI,TASKB_INFO

;
MOV     AX,CS           ; セグメントをセット
MOV     [SI],_CS,AX
MOV     [DI],_CS,AX
MOV     AX,DS
MOV     [SI],_DS,AX
MOV     [DI],_DS,AX
MOV     AX,ES
MOV     [SI],_ES,AX
MOV     [DI],_ES,AX
LEA     AX,BEGINA       ; プログラム実行開始アドレスをセット
MOV     [SI],_IP,AX
LEA     AX,BEGINB
MOV     [DI],_IP,AX
MOV     AX,100H         ; フラグをセット (TFのみON)
MOV     [SI],_FLAGS,AX
MOV     [DI],_FLAGS,AX

;
LEA     AX,TASKA_INFO   ; 最初に実行するタスクのアドレスをセット
MOV     CUR_TASK,AX
PUSHF                                ; TFをセット
POP
OR      AX,100H
PUSH    AX
POPF

;
BEGINA PROC                ; タスクA
MOV     AX,0A000H        ; テキストVRAMのセグメント
MOV     ES,AX
XOR     DI,DI            ; タスクAは左から
MOV     CX,160/2*25      ; 画面1行は160バイト,25行分

;
BEGINA_LOOP:
PUSH    CX
XOR     ES:WORD PTR [DI],0FFFFH ; 画面へ書き込み
MOV     CX,10
$
LOOP    $
ADD     DI,2
POP     CX
LOOP    BEGINA_LOOP

;
MOV     AX,4C00H
INT     21H

;
BEGINA ENDP

;
BEGINB PROC                ; タスクB
MOV     AX,0A000H        ; テキストVRAMのセグメント
MOV     ES,AX
MOV     DI,160*25-2      ; タスクBは右下から
MOV     CX,160/2*25      ; 画面1行は160バイト,25行分

;
BEGINB_LOOP:
PUSH    CX
XOR     ES:WORD PTR [DI],0FFFFH ; 画面へ書き込み
MOV     CX,10
LOOP    $
SUB     DI,2
POP     CX
LOOP    BEGINB_LOOP

;
MOV     AX,4C00H

```

```

;          INT      21H
;
; BEGINB  ENDP
;
; SINGLESTEP      PROC      ; シングルステップ割り込み処理
;                   ASSUME   CS:CODE,DS:NOTHING,ES:NOTHING,SS:NOTHING
;                   STI      ; 割り込みを禁止する
;                   PUSH     AX
;                   PUSH     BX
;                   PUSH     CX
;                   PUSH     DX
;                   PUSH     SI
;                   PUSH     DI
;                   PUSH     BP
;                   PUSH     DS
;                   PUSH     ES
;                   MOV      BP,SP
;
;                   CMP      CUR_TASK,OFFSET TASKA_INFO      ; 今のタスクを調べる
;                   LEA      SI,TASKA_INFO
;                   LEA      DI,TASKB_INFO
;                   JE        SAVE_INFO      ; タスクAの情報を待避
;
;                   XCHG     SI,DI
;
; SAVE_INFO:      ; 直前のタスクの情報を待避
;                   MOV      AX,[BP]      ; 各レジスタの内容を待避
;                   MOV      CS:[SI]_ES,AX
;                   MOV      AX,[BP+2]
;                   MOV      CS:[SI]_DS,AX
;                   MOV      AX,[BP+4]
;                   MOV      CS:[SI]_BP,AX
;                   MOV      AX,[BP+6]
;                   MOV      CS:[SI]_DI,AX
;                   MOV      AX,[BP+8]
;                   MOV      CS:[SI]_SI,AX
;                   MOV      AX,[BP+10]
;                   MOV      CS:[SI]_DX,AX
;                   MOV      AX,[BP+12]
;                   MOV      CS:[SI]_CX,AX
;                   MOV      AX,[BP+14]
;                   MOV      CS:[SI]_BX,AX
;                   MOV      AX,[BP+16]
;                   MOV      CS:[SI]_AX,AX
;                   MOV      AX,[BP+18]
;                   MOV      CS:[SI]_IP,AX
;                   MOV      AX,[BP+20]
;                   MOV      CS:[SI]_CS,AX
;                   MOV      AX,[BP+22]
;                   MOV      CS:[SI]_FLAGS,AX
;
; RECOVER_INFO:   ; 次のタスクに対して情報を提供
;                   MOV      AX,CS:[DI]_ES      ; 各レジスタの内容を次のタスクに対して復帰
;                   MOV      [BP],AX
;                   MOV      AX,CS:[DI]_DS
;                   MOV      [BP+2],AX
;                   MOV      AX,CS:[DI]_BP
;                   MOV      [BP+4],AX
;                   MOV      AX,CS:[DI]_DI
;                   MOV      [BP+6],AX
;                   MOV      AX,CS:[DI]_SI
;                   MOV      [BP+8],AX
;                   MOV      AX,CS:[DI]_DX
;                   MOV      [BP+10],AX
;                   MOV      AX,CS:[DI]_CX
;                   MOV      [BP+12],AX
;                   MOV      AX,CS:[DI]_BX
;                   MOV      [BP+14],AX

```



```

MOV     AX,CS:[DI],_AX
MOV     [BP+16],AX
MOV     AX,CS:[DI],_IP
MOV     [BP+16],AX
MOV     AX,CS:[DI],_CS
MOV     [BP+20],AX
MOV     AX,CS:[DI],_FLAGS
MOV     [BP+22],AX
;
MOV     CS:CUR_TASK,DI      ; 現在のタスクをセット
POP     ES                  ; 各レジスタの復帰とジャンプ
POP     DS
POP     BP
POP     DI
POP     SI
POP     DX
POP     CX
POP     BX
POP     AX
IRET
SINGLESTEP  ENDP
;
OPENING DB      13,10
DB      'このプログラムは2個の処理の並行実行のサンプルです。'
DB      13,10,'$'
;
SINGLE  ENDP
;
CODE   ENDS
;
END     SINGLE

```

■図 7.1 SINGLE.COM ダンプリスト

```

00000000 : EB 33 90 00 00 00 00 00 00 00 00 00 00 00 00 : 1AE
00000010 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 : 000
00000020 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 : 000
00000030 : 00 00 00 00 00 00 8D 16 A1 02 84 09 CD 21 84 25 80 : 47A
00000040 : 01 8D 16 CD 01 CD 21 8D 36 05 01 8D 3E 1D 01 8C : 49E
00000050 : C8 89 44 10 89 45 10 8C D8 89 44 12 89 45 12 8C : 632
00000060 : C0 89 44 14 89 45 14 8D 06 8C 01 89 44 0E 8D 06 : 511
00000070 : AC 01 89 45 0E 88 00 01 89 44 16 89 45 16 8D 06 : 49C
00000080 : 05 01 A3 03 01 9C 58 0D 00 01 50 9D 88 00 A0 8E : 482
00000090 : C0 33 FF 89 D0 07 51 26 81 35 FF FF 89 0A 00 E2 : 852
000000A0 : FE 83 C7 02 59 E2 EF 88 00 4C CD 21 88 00 A0 8E : 84C
000000B0 : C0 BF 9E 0F 89 D0 07 51 26 81 35 FF FF 89 0A 00 : 7AA
000000C0 : E2 FE 83 EF 02 59 E2 EF 88 00 4C CD 21 F8 50 53 : 90E
000000D0 : 51 52 56 57 55 1E 06 88 EC 2E 81 3E 03 01 05 01 : 437
000000E0 : 8D 36 05 01 8D 3E 1D 01 74 02 87 F7 88 46 00 2E : 4A5
000000F0 : 89 44 14 88 46 02 2E 89 44 12 88 46 04 2E 89 44 : 491
00000100 : 0C 88 46 06 2E 89 44 0A 88 46 08 2E 89 44 08 88 : 44F
00000110 : 46 0A 2E 89 44 06 88 46 0C 2E 89 44 04 88 46 0E : 40C
00000120 : 2E 89 44 02 88 46 10 2E 89 04 88 46 12 2E 89 44 : 477
00000130 : 0E 88 46 14 2E 89 44 10 88 46 16 2E 89 44 16 2E : 424
00000140 : 88 45 14 89 46 00 2E 88 45 12 89 46 02 2E 88 45 : 492
00000150 : 0C 89 46 04 2E 88 45 0A 89 46 06 2E 88 45 08 89 : 44B
00000160 : 46 08 2E 88 45 06 89 46 0A 2E 88 45 04 89 46 0C : 408
00000170 : 2E 88 45 02 89 46 0E 2E 88 05 89 46 10 2E 88 45 : 478
00000180 : 0E 89 46 12 2E 88 45 10 89 46 14 2E 88 45 16 89 : 47D
00000190 : 46 16 2E 89 3E 03 01 07 1F 5D 5F 5E 5A 59 58 58 : 3FB
000001A0 : CF 0D 0A 82 B1 82 CC 83 76 83 8D 83 4F 83 89 83 : 7D1
000001B0 : 80 82 CD 32 8C C2 82 CC 8F 88 97 9D 82 CC 95 C0 : 988
000001C0 : 8D 73 8E C0 8D 73 82 CC 83 54 83 93 83 76 83 88 : 890
000001D0 : 82 C5 82 B7 81 44 0D 0A 24 : 380

```

## 7.2 裏で本物を走らせる

### ○考え方

マルチタスクの考え方に似ていますが、今回はシングルステップ割り込みによって行われる処理が本物で、表で実行されるプログラムは偽物であるというものです。

### ○実現方法

最も簡単なのが、シングルステップ割り込み処理ルーチンを複数個用意するというものです。すなわち、シングルステップ割り込み処理ルーチンに、番号を①から付けるならば、1回目のシングルステップ割り込みで①が実行され、①には次に②が実行されるように、割り込みベクタを書き換えておきます。よって、次は②が実行されますが、②では次に③が実行されるように、同様に割り込みベクタを書き換えておきます。これを必要な回数だけ繰り返して、次々とプログラムが実行されるわけです。このとき、表では何か意味ありげな処理を行わせてカモフラージュします。

### ○サンプル

裏で表のプログラムを復元し動作させる、サンプル SINGLE2.COM を、実行例と共に図 7.2 として示します。

■図 7.2 SINGLE2.ASM ソースリスト

```

;
;*****
;
;      SINGLE2.ASM
;
;      シングルステップ割り込みを、裏のタスクとするサンプル
;
;      裏のタスクは、表のタスクの暗号化を解きます。
;
;      COPYRIGHT(C) 1987 BY SHUWA SYSTEM TRADING CO.,LTD.
;
;      LAST MODIFIED ON FEBRUARY 5TH,1987
;
;*****
;
CODE    SEGMENT
        ASSUME    CS:CODE,DS:CODE,ES:CODE,SS:CODE
;
;      ORG        100H
;
SINGLE2  PROC
        JMP        MAIN
;
CUR_ADDRESS    DW        ?           ; 次に暗号化を解除するアドレス
;
MAIN:
        LEA        DX,OPENING        ; 開始メッセージの表示
        MOV        AH,9
        INT        21H
;
        LEA        SI,TOP
        MOV        DI,SI
        MOV        CX,256
        CLD
;
SECRET:
        LODSB
        XOR        AL,0AAH           ; 排他的論理和による暗号化
        STOSB
        LOOP       SECRET
;
        MOV        AH,25H           ; シングルステップ割り込みのベクタを変更する
        MOV        AL,1
        LEA        DX,SINGLESTEP
        INT        21H
;
        LEA        AX,TOP            ; 暗号化解除アドレスを設定
        MOV        CUR_ADDRESS,AX   ; シングルステップ割り込みをアクティブにする
        PUSHF
        POP        AX
        OR         AX,100H
        PUSH       AX
        POPF
        NOP
        NOP
        JMP        TOP
;
;      ORG        200H
;
TOP:
;      ; ここから256バイトは、暗号化されます。
        LEA        DX,MESSAGE1
        MOV        AH,9
        INT        21H
;
        LEA        DX,PROMPT1       ; 文字列を入力
        MOV        AH,9

```

```

INT     21H
LEA     DX,BUFFER1
MOV     AH,10
INT     21H
;
LEA     DX,PROMPT2      ; 文字列を入力
MOV     AH,9
INT     21H
LEA     DX,BUFFER2
MOV     AH,10
INT     21H
;
LEA     SI,BUFFER1+2    ; 文字列を連結
LEA     DI,BUFFER3+2
MOV     CL,BUFFER1+1
XOR     CH,CH
REP     MOVSB
LEA     SI,BUFFER2+2
MOV     CL,BUFFER2+1
REP     MOVSB
;
MOV     AL,'$'
STOSB
LEA     DX,BUFFER3      ; 連結した文字列を表示
MOV     AH,9
INT     21H
;
MOV     AX,4C00H        ; プログラム終了
INT     21H
;
ORG     300H            ; 破壊を逃れるため
;
SINGLESTEP PROC
PUSH    AX
PUSH    BX
PUSH    CX
;
MOV     BX,CS:CUR_ADDRESS ; 暗号化を解除する
CMP     BX,OFFSET TOP+256 ; すべて暗号を解除したか?
JAE     SINGLESTEP_EXIT ; 解除したなら終わり
;
MOV     CX,8             ; 暗号を8バイト解除する
;
SINGLESTEP_LOOP:
MOV     AL,CS:[BX]
XOR     AL,0AAH
MOV     CS:[BX],AL
INC     BX
LOOP    SINGLESTEP_LOOP
;
SINGLESTEP_EXIT:
MOV     CS:CUR_ADDRESS,BX ; 次に解除するアドレスをセット
;
POP     CX
POP     BX
POP     AX
IRET
SINGLESTEP ENDP
;
OPENING DB 13,10
DB 'このプログラムはシングルステップ割り込みを裏の処理に'
DB '用いたサンプルです。'
DB 13,10,'$'
;
MESSAGE1 DB '暗号化されているはずの部分を実行しています。'
DB 13,10,'$'
;

```

```

PROMPT1 DB      13,10
          DB      '文字列を連結します。1番目の文字列を入力して下さい：'
          DB      '$'
;
PROMPT2 DB      13,10
          DB      '2番目の文字列を入力して下さい：'
          DB      '$'
;
BUFFER1 DB      16,?
          DB      16 DUP(?)
;
BUFFER2 DB      16,?
          DB      16 DUP(?)
;
BUFFER3 DB      13,10
          DB      32 DUP(?)
;
SINGLE2 ENDP
;
CODE      ENDS
;
END        SINGLE2

```

■図 7.2 SINGLE2.COM ダンプリスト

```

00000000 : EB 03 90 00 00 8D 16 25 03 B4 09 CD 21 8D 36 00 : 4B7
00000010 : 02 88 FE 89 00 01 FC AC 34 AA AA E2 FA 84 25 80 : 8DA
00000020 : 01 8D 16 00 03 CD 21 8D 06 00 02 A3 03 01 9C 58 : 3C5
00000030 : 0D 00 01 50 9D 90 90 90 E9 C5 00 00 00 00 00 00 : 459

```

0040Hから00FFHまですべて00H

```

00000100 : 8D 16 72 03 B4 09 CD 21 8D 16 A1 03 B4 09 CD 21 : 5B5
00000110 : 8D 16 F8 03 B4 0A CD 21 8D 16 D8 03 B4 09 CD 21 : 676
00000120 : 8D 16 0D 04 B4 0A CD 21 8D 36 FD 03 8D 3E 21 04 : 513
00000130 : 8A 0E FC 03 32 ED F3 A4 8D 36 0F 04 8A 0E 0E 04 : 5CD
00000140 : F3 A4 8D 24 AA 8D 16 1F 04 B4 09 CD 21 88 00 4C : 68A
00000150 : CD 21 00 00 00 00 00 00 00 00 00 00 00 00 00 : 0EE

```

0160Hから01FFHまですべて00H

```

00000200 : 50 53 51 2E 8B 1E 03 01 81 FB 00 03 73 0E B9 08 : 490
00000210 : 00 2E 8A 07 34 AA 2E 88 07 43 E2 F5 2E 89 1E 03 : 54C
00000220 : 01 59 58 58 CF 0D 0A 82 81 82 CC 83 76 83 8D 83 : 700
00000230 : 4F 83 89 83 80 82 CD 83 56 83 93 83 4F 83 8B 83 : 7FF
00000240 : 58 83 65 83 62 83 76 8A 84 82 E8 8D 9E 82 0D 82 : 8A2
00000250 : F0 97 A0 82 CC 8F 88 97 9D 82 C9 97 70 82 A2 82 : 988
00000260 : 8D 83 54 83 93 83 76 83 88 82 C5 82 B7 81 44 0D : 803
00000270 : 0A 24 88 C3 8D 86 89 8B 82 83 82 EA 82 C4 82 A2 : 8DB
00000280 : 82 E9 82 CD 82 B8 82 CC 95 94 95 AA 82 F0 8E 0C : A6A
00000290 : 8D 73 82 B5 82 C4 82 A2 82 DC 82 B7 81 44 0D 0A : 814
000002A0 : 24 0D 0A 95 B6 8E 9A 97 F1 82 F0 98 41 8C 8B 82 : 81A
000002B0 : 85 82 DC 82 B7 81 44 82 50 94 D4 96 DA 82 CC 95 : 99E
000002C0 : 86 8E 9A 97 F1 82 F0 93 FC 97 CD 82 B5 82 C4 89 : AD1
000002D0 : 8A 82 B3 82 A2 3A 20 24 0D 0A 82 51 94 D4 96 DA : 753
000002E0 : 82 CC 95 B6 8E 9A 97 F1 82 F0 93 FC 97 CD 82 B5 : AE5
000002F0 : 82 C4 89 BA 82 B3 82 A2 3A 20 24 10 00 00 00 00 : 570
00000300 : 00 00 00 00 00 00 00 00 00 00 00 00 10 00 00 : 010
00000310 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0D : 00D
00000320 : 0A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 : 00A

```

## 資料編





1. CPU
  2. OS
  3. マシン
- 

資料編では本書を読み進むにあたって、説明しておいたほうがよいと思われる知識についてまとめてあります。構成はCPU関連、OS関連、用語関連としてあります。なお本書においては、資料編を膨大にさせるのは目的に合わないため、説明は必要最小限に留めてあります。手元に適当な参考書を置いて、本書を読まれることをお勧めします。



# 1

## CPU

ソフトウェアによるプロテクトは、プログラム自体がプロテクトともいえるのですから、それを破るにはプログラム=CPU（命令セット）についての知識が不可欠です。ここでは CPU についての資料から解説してゆきます。

## 1.1 搭載されるCPU

PC-9800 シリーズ本体（PC-9801/E/F/M/U/VF/VM/UV/VX）のメイン CPU は機種ごとに、次のように異なった種類のものが積まれています。

### ■ 8086

PC-9801/E/F/M に積まれているのが 8086 です（正確には i8086 のセカンドソース、 $\mu$ PD8086）。ただし、PC-9801 に積まれているのはクロック周波数 5MHz のもので、その他の機種はクロック周波数 8MHz のものが積まれています。兄弟プロセッサに 8088 があり、こちらは IBM-PC 等に積まれています。



## ■V30

PC-9801U/VF/VM/UV/VX に積まれているのが通称 V30 と呼ばれる日本電気オリジナルの CPU です（正確には  $\mu$ PD80286）。ただし、PC-9801U に積まれているのはクロック周波数 8MHz のもので、その他の機種はクロック周波数 10MHz のものが積まれています。兄弟プロセッサに V20 があります。

## ■80286

PC-9801VX に積まれているのが 80286 です（正確には i80286 相当品）です。クロック周波数は 8MHz です。80286 には、プロテクションモードと 80286 モードと呼ばれる 2 つのモードがありますが、ここで取り扱うのは 8086 とほぼ動作が同じ 8086 モードです。

プロテクションモードは広大なメモリを必要とし、また保護機構が必要な OS(PC-UX など)で使用されます。

V30 と 80286 は、8086 を含むかたちで設計されたために、8086 用に作成されたプログラムであればほぼ完全に動作します。ここで“ほぼ完全に”と書いたのは、V30/80286(8086 モード)と 8086 には、その命令とアーキテクチャに微妙な差があり、細かな箇所では同じ動作をするとは限らないからです。たとえば V30 と 80286 には拡張命令があり、8086 にはない便利な命令を使用することができます（ビット操作、スタック操作命令など）。また 8086 のシフト命令と 80286 のシフト命令は、表向きは同じ命令でも細かな点で微妙な差があります。8086 で使える命令で、80286 では使えない命令があります。

また、決定的なのは CPU 自体のパフォーマンスです。V30 や 80286 では非常に高速に動作するため、周辺機器とのタイミングが合わないということも起こり得ます。たとえば(株)管理工学研究所の『松』ですが、V30 の 10MHz モードでは動作しません。8MHz に落として初めて動作します。これは『松』が GDC に対して直接書き込みを行っているためで、『松』は 8086 に対応するかたちでプログラミングされているため、V30 では速すぎたのでしょう

一般に周辺 LSI に書き込みを行った際、あるものは一定の待ち時間を設ける必要がありますが、これはソフトウェアでタイミングをとっているからです。

このような、CPU が異なることによる特性は、プロテクトに使用する上で役に立つことがあります。特に、機種を限定するようなプロテクトには最適です（実際の例については応用編を参照）。

## 1.2 メモリ管理

ここでは、8086 のメモリの扱いについて触れます。

### ■セグメント

8086 では、セグメントというメモリ管理方法を用いています。これは 8086 の持つ 1MB のアドレス空間を、64KB ごとに区切って管理するというもので、8085 などの、8 ビット CPU からの移行を容易にするというものです。

8086 では、すべてのレジスタは最も長いものでも 16 ビット長です。16 ビットでは最大 64 (=2<sup>16</sup>) KB のメモリしか指定することができません。これでは 1MB のアドレス空間をカバーすることが

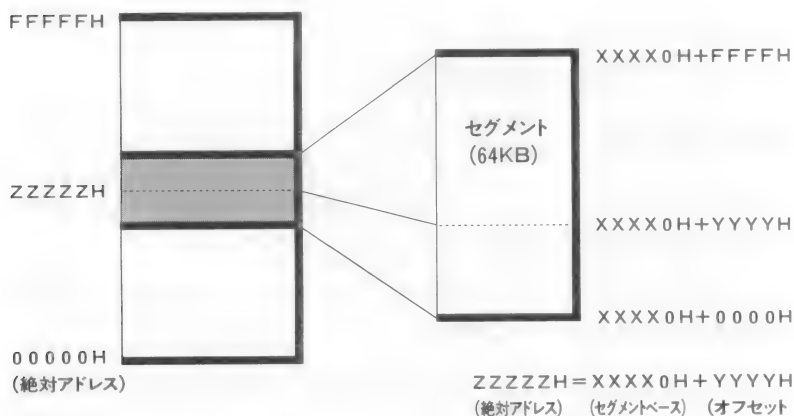
できませんので、セグメントベースという 64KB のメモリ空間のベースになる値を用意し、それを変化させることで、結果的に 1MB のアドレス空間をカバーしようというものです。

セグメントベースは 16 バイトおき（パラグラフ単位といいます）にとることができます。セグメントベースも、実体は 16 ビット長のレジスタであり、1MB をカバーするために必要な 20 ビットに合わせるために、アドレス計算時には 4 ビット左へシフトする（16 倍することと等価）という演算を施しています。これが、セグメントベースが 16 バイト＝4 ビットおきにしか配置できない理由です。

セグメントベースに対してそこからの距離はオフセットと呼ばれ、実際のアドレスの計算は以下のように行われます（図 1.1 参照）。

- ①セグメントベースを左へ 4 ビットシフト（16 倍する）
- ②オフセットアドレスを加える

■図 1.1 セグメント

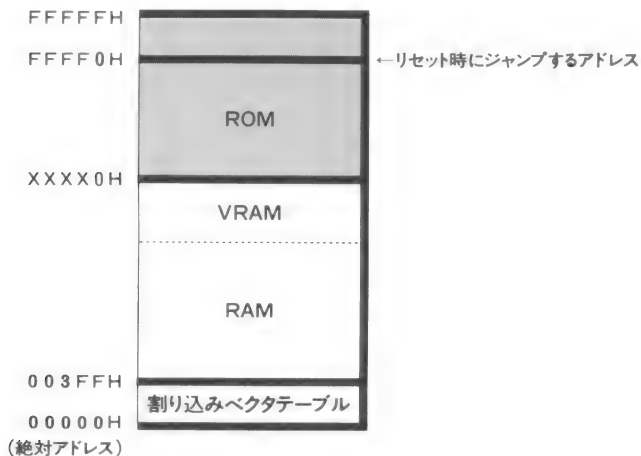


## ■標準的なメモリレイアウト

8086 には、標準的なメモリレイアウトというものがあります。それは、CPU の特性によって決められるものであり、オペレーティングシステムなどでは、それをふまえた上で動作することが要求されています（図 1.2 参照）。

図 1.2 では、ROM にあたる部分は網をかけてあります。1MB のアドレス空間において、もっとも低位には割り込みベクタテーブルが位置しています（00000H～003FFH）。この位置は固定されており、ユーザが自由に動かしたりすることはできません。しかし、続く領域はユーザが自由に使用することができます。ここまでが RAM で、RAM の最上位には VRAM が置かれたりもします。

■図 1.2 標準的なメモリレイアウト



RAMの上にはROMがきます。基本的にRAMは低位に、ROMは高位に位置するのがふつうで、これは8086のリセット時のジャンプアドレス（FFFFH:0000H,FFFF0H）に依っているのです。

もっと細かなメモリレイアウトは、OSによって決定されます。これについては後述します。

## 1.3 レジスタ

レジスタとは、CPU内部で一時的な、もしくは固定された目的のデータを保存するためのものです。ここでは、8086の持つレジスタについて触れましょう。

### ■汎用レジスタ

8086には、AX, BX, CX, DX, SI, DI, BP, SPと呼ばれる汎用的なレジスタが用意されています（すべて16ビット長）。このうち、前の4本はxH,xLという8ビットレジスタに分解できます（xはA, B, C, Dのいずれか。AXはAH, ALという具合に）。あとの4本は分割することができません。

汎用レジスタはメモリを指し示したり、データの保管に用いたり、各種の演算に用いたりする。最も使用頻度の高いレジスタ群です。

各レジスタの主な用途をまとめておきます。

**A X** アキュムレータ。あらゆる転送、演算において優位（実行時間が短い、命令コードが短いなど）に位置します。また、アキュムレータのみを対象とした命令も存在することに注意してください。

- B X** ベースポインタ。メモリを指す際のベースとして用いられます。
- C X** カウンタレジスタ。ループカウンタやシフトカウンタに用いられます（シフトカウンタは **C L** のみ）。
- D X** データレジスタ。データの保持用や乗除算命令における補助レジスタとして用いられます。
- S I** ソースインデックス。メモリ転送の際のソースとして用いられます。
- D I** ディスティネーションインデックス。メモリ転送の際のディスティネーションとして用いられます。
- B P** ベースポインタ。**B X** と同様にメモリを指す際のベースとして用いられますが、セグメントベースを、後述する **SS** とすることに注意してください。
- S P** スタックポインタ。スタックの位置を指します。用途が限定されています。

これら汎用レジスタは、推奨される目的から外れた使い方をされるなど、かなりトリッキーな使い方があります。

また、同様に推奨される目的があることを知らずに、プログラミングされている場合もあります。注意が必要です。

## ■セグメントレジスタ

セグメントレジスタは、前述のセグメントベースを保持する目的で用いられるレジスタ群です。名称は **CS,DS,ES,SS** となっており、命令コードを取り出すセグメント（コードセグメント、レジスタ **CS**）、データを保持するセグメント（データセグメント、レジスタ **DS, ES**）、スタックに用いるセグメント（スタックセグメント、

レジスタ SS) のベースをそれぞれ保持します。セグメントレジスタは、ふつうに使用する限り、あまり操作することのないレジスタです。しかし、どの時点でどのセグメントレジスタがどのような値を持つのかは、常に把握している必要があります。そうでないと、間違った位置からデータを取り出したり、また格納してしまったりすることにもなるからです。

## ■特殊なレジスタ

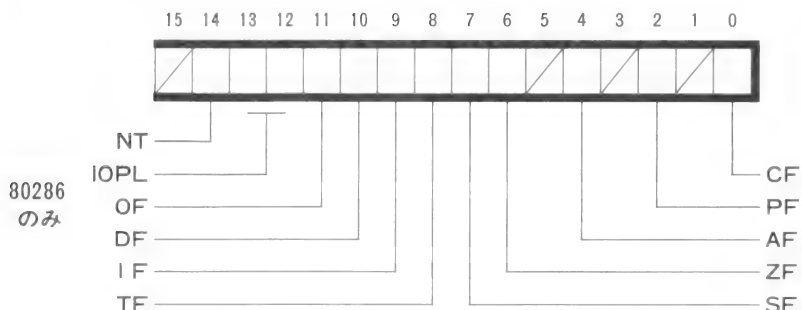
最後は、インストラクションポインタ（レジスタ IP）と呼ばれる命令実行位置を保持するレジスタと、フラグレジスタと呼ばれるフラグの集合体について説明します。インストラクションポインタは、レジスタ CS をセグメントベースとして、その中での命令取り出し位置を保持するものです。命令は、レジスタ CS:IP の組から絶対アドレスを求められ、そこから取り出されます。命令が取り出されると、命令の長さだけレジスタ IP の内容が増加します。

フラグというのは、各種演算などにおいてその状態が変化し、続く命令で演算結果など、参照できるようにするためのものですが、フラグの内容をまとめて扱えるよう、16 ビットのレジスタとしてまとめられています。フラグレジスタの構成は図 1.3 のようになっています。

それぞれのフラグは、以下のような意味を持っています。

- CF** 演算の結果、最上位ビットからの桁上げ、または最上位ビットへの桁借りが生じた場合に 1 となります。エラー発生の情報などを返す目的でも使用されます。
- AF** 演算の結果、下位から上位への桁上げ、または上位から下位への桁借りが生じた場合に 1 となります。算術補正命令

■図 1.3 フラグレジスタ



で参照されることのみで、外部で参照することはほとんどありません。

- ZF** 演算結果が 0 となった場合に 1 となります。
- SF** 演算結果が負となった場合に 1 となります。
- OF** 演算の結果、オーバフローが生じた場合に 1 となります。  
INTO 命令で暗黙に参照されます。
- PF** 演算の結果、1 であるビットが偶数個ある場合に 1 となります。あまり顔を出しません。
- TF** シングルステップ割り込みを発生させたいときに 1 とします。これを設定する命令は特に存在しません。
- IF** 割り込みをマスクしたいときに 1 とします。ただし NMI（マスク不能割り込み）はマスクすることはできません。
- DF** スtring 命令において、アドレス増減方向を減としたいときに 1 とします。

このうち、CF, AF, ZF, SF, OF, PF の 6 個は、演算の結果において自動的に設定されますが、あとの TF, IF, DF の 3 個は、要求に応じて設定してやる必要があります。



## 1.4 アドレッシングモード

アドレッシングモードとは、データをアクセスする方法を指定するものです。8086 ではこれが多岐にわたっているのが特徴です。

### ■レジスタモード

レジスタとレジスタの間で転送を行うモードです。CPU 内部で処理されるため、実行が非常に高速に行えるのが特徴です。

<例>   MOV     AX,BX

### ■イミディエイトモード

定数を扱うモードです。定数をレジスタ、あるいはメモリに転送する際に用いますが、定数自体は命令コードに含まれているため、そのぶん実行は高速です。

<例>   MOV     AX,1000H

### ■メモリモード

メモリに対して転送を行うモードです。このとき、メモリを指し示す方法はいろいろあり非常に複雑です。一般にアドレッシングモードといえはメモリモードを指し、メモリを指し示す際に使用できるレジスタは、

BX, BP, SI, DI

であって、定数も用いることができます。またレジスタとレジスタ、レジスタと定数、レジスタ・レジスタと定数、という組み合わせでアドレスを指定することもできます。許される組み合わせは、次のようになっています。

[定数]

[BX]

[BX+定数]

[BX+SI]

[BX+SI+定数]

ここで BX は BP に、SI は DI に置き換えることができます。一般にメモリをアクセスする際のセグメントベースは、レジスタ DS が採用されます。ただし、例外がありレジスタ BP を用いた場合には、レジスタ SS がセグメントベースとして採用されます。

また、レジスタ SP とレジスタ IP も、メモリを指し示すのに用いることができますが、これらは用途が固定されており、レジスタ SP が、レジスタ SS をセグメントベースとしたスタック保持用、レジスタ IP が、レジスタ CS をセグメントベースとした命令ポイント保持用として用いられています。PUSH, POP 命令では、暗黙のうちにレジスタ SP が参照され、JMP 命令などでは、暗黙のうちにレジスタ IP が参照されている、と考えることができます。

## 1.5 その他のことから

そのほかに、CPU に関して知っておいたほうがよいことを列記します。

### ■リセット後の動作

8086 では、ハードウェアリセットが入るとセグメント **FFFFH**、オフセット **0000H** で示されるアドレスへ自動的にジャンプします。ここに、マシンの初期化プログラムへのジャンプ命令を置いておくのがふつうです。

この時点で、レジスタ **CS** を除くすべてのレジスタは、**0000H** を値として持つように初期化されます。レジスタ **CS** は **0FFFFH** となります。またフラグはすべてリセットされ、後述するプリフェッチキューは空になります。

### ■プリフェッチキュー

8086 には、命令の連続実行をスムーズに行うため、プリフェッチキューというものを用意して、ひまを見ては命令を前もって読み込んでおくという機構が備わっています。これは、除算命令などの多くの時間を要する命令において、演算中に、あらかじめ続く命令を取り込んでおくというもので、次に命令を実行するとき、改めて命令を読み込む必要がなくなり、効率のよい命令実行が可能になります。

プリフェッチキューの特性を応用すると、極めて高度なプロテクトが可能になります。具体的な内容については、応用編を参照してください。

## ■セグメントレジスタへの書き込み

8086 には、セグメントレジスタへの書き込みを行った際、続く 1 命令が実行終了するまでは、割り込みが禁止されるという機構も備わっています。これはスタックの設定を 1 命令で行うことができないという、8086 の命令セットからいえば当然のことですが、具体的な例をあげれば、次のようなことになります。

```
MOV  SS,STACK_SEG
MOV  SP,OFFSET STACK_BOTTOM
```

ここで 1 番目の命令では、スタックのセグメントがレジスタ SS に設定され、スタックセグメントの移動は完了しますが、レジスタ SP の値が不定であるため、メモリのない場所にスタックが設定されてしまうこともあり得ます。このようなときに割り込みが入れば、ほぼ 100% の確率で暴走するでしょう。

8086 では、このようなことを防ぐために、続くレジスタ SP の設定が終了するまでは、割り込みを受け付けません。

## ■割り込みベクタテーブル

メモリレイアウトの項でも触れたように、00000H から 003FFH の領域は割り込みベクタテーブルとして予約されています。割り込みベクタテーブルは、00H から FFH のタイプを持つ割り込みに対

して、ジャンプ先のアドレスを並べたもので、図 1.4 のような構造を持ちます。

■図 1.4 割り込みベクタテーブル

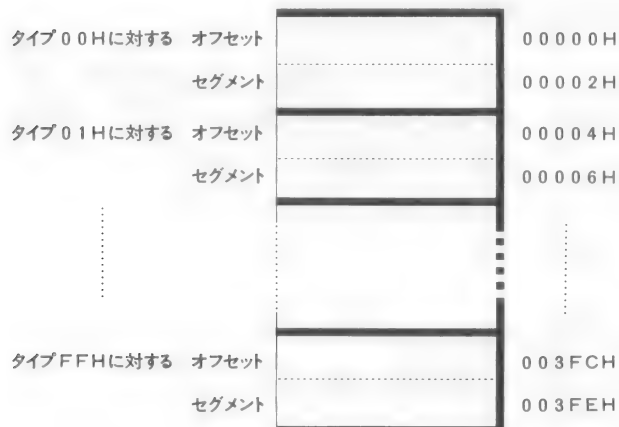


図 1.4 からわかるように、1 個の割り込みに対して 4 バイトの領域が確保されています。また順番としてはオフセット、セグメントの順となっています。割り込みのタイプとアドレスは 1 対 1 になっていますから、INT 命令を用いずに、割り込み処理ルーチンを呼び出すことも容易です。

## 2

## OS

現在、多くのアプリケーションはMS-DOS、DISK BASICといったOS（オペレーティングシステム）と呼ばれるプログラム上で動作します。中には、ゲームなどのようにOSを持たないものもありますが、ビジネス用の多くのアプリケーションは、MS-DOSかDISK BASICの上で動作します。プログラムを読むためには、CPU自体の知識のほかにOSの知識も重要です。

ここでは、OSについてのことがらを資料としてまとめておきます。

## 2.1 OS起動のメカニズム

OSを持たないゲームなどはどこから解読すればよいか、OS上で動作するソフトはどこから解読すればよいか、そのための知識を解説します。

### ■リセットイニシャル

本編1において、8086CPUはリセット時にFFFFH:0000Hで示されるアドレスへジャンプすると書きましたが、ここにはふつうマシンの初期化プログラムへの、セグメント外ジャンプ命令が書かれています。

初期化プログラムではメモリのチェック、周辺 LSI などの初期化とワークエリアへの値の設定などを行った後、ブートストラップローダと呼ばれるルーチンがコールされます。

## ■ブートストラップローダ

プログラムの初期化が終了すると、ブートストラップローダがコールされ、ディスクドライブの接続状況に合わせて、IPL（イニシャルプログラムローダ）と呼ばれるプログラムをディスク上から読み込みます。IPL をディスク上から引っ張り上げるという動作から、ブートストラップという名称が付いているのです（ブートストラップとはブーツに付いているペラのことです）。

ブートストラップローダは、接続されているディスクドライブを順にチェックしますが、このとき優先順位というものが決められており、PC-9801 ではメモリスイッチでこれを決定します。ふつうは 1MB ディスク（両用タイプ）、640KB ディスク、1MB ディスク（両用タイプでないもの）、320KB ディスク、ハードディスクの順となっており、ディスクの挿入されていないドライブや、読み込みに失敗したドライブはスキップされ、最も早く見つけられた異常のないドライブから、IPL がロードされます。

ブートストラップローダはシステムにその位置が固定されており、ここをコールすることで、IPL の再読み出しを行うことも可能です。ブートストラップローダの位置と呼び出し法は、以下のようになっています。

位置： FD80H：27E8H

呼び出し法： セグメント外CALL

呼び出し時の条件： DS←0000H

AL ← ブートプライオリティ初期値

AH ← ブートプライオリティ終了値

ブートストラップローダは、必ずセグメント外CALLによって行われなければなりません。また、レジスタの内容は保存されませんので、必要ならば呼び出し時に待避しておかなければなりません。ブートストラップローダが呼び出し可能であるということは、プログラム解析の大きな助けとなります。詳細は後述します。

ブートプライオリティとは、ブート対象のドライブと優先順位を指定するもので、ドライブの種類によって、以下の値を持ちます（値の小さいほうが優先順位は高い）。

- 01H: 1MBディスク（両用タイプのもの）
- 02H: 640KBディスク
- 04H: 1MBディスク（両用タイプでないもの）
- 08H: 320KBディスク
- 0AH: 5"HD（#0）
- 0BH: 5"HD（#1）

ブートプライオリティの初期値と終了値を、それぞれ04H,08Hとした場合、1MB 両用ディスク、640KB ディスク、ハードディスクへの読み出しは行われなくなり、かつ1MB ディスク（両用でないもの）のほうに読み出しが試みられます。

なお、ブートストラップローダ呼び出し時には、INT 18H～INT 1BH に対する割り込みベクタを、最小限 ROM BASIC と同じものにしておく必要があります。内容は機種によって異なりますのであらかじめ調べておいてください。



## ■IPL (イニシャルプログラムローダ)

IPL は、ブートストラップローダによってディスク上から読み出され、メモリ上の 1FC00H (1MB, 640KB ディスク) あるいは 1FE00H (その他) の領域に読み出されます。読み出される量は前者で 1024 バイト、後者で 512 バイトです。正常な IPL の読み出しを期待するには、表 2.1 に示す IPL の形式を守る必要があります。

■表 2.1 IPL の形式

立上げ可能 装置番号	記録密度	セクタ長	セクタ数	バイト数	IPL ロード アドレス	採用 OS
0, 1, 2, 3	単密度	128	4	512	1FE0H : 0000H	BASIC (1MB ディスク)
		256	2			
		512	1			
	倍密度	256	4	1024	1FC0H : 0000H	CP/M, BASIC (640KB ディスク)
		512	2			MS-DOS (640KB ディスク)
		1024	1			MS-DOS (1MB ディスク)

IPL は、そのディスクの OS を決定するもので、DISK BASIC, MS-DOS などで固有の内容となっています。IPL によってその OS に固有のシステムコードがさらにロードされますので、実際の解析は IPL から行えばよいことになります。ただし、MS-DOS などの OS 上で動作するソフトなどは、IPL から解析することが無駄な場合も多いようです。

## 2.2 MS-DOSの構造

現在、MS-DOSはPC-9800シリーズを代表するOSです。MS-DOSの上では、各種ワープロソフト、データベースソフト、表計算ソフト、作画ソフトなど、多くのソフトウェアが動作していますから、プロテクト解析の際の、最も大きなターゲットといえます。そこでOS個別の解説として、まずは、MS-DOSから解説してゆきます。

### ■構成するパーツ

MS-DOSは、主にIO.SYS、MSDOS.SYS、COMMAND.COMという3つのパーツから構成されます。このうちで、IO.SYSとMSDOS.SYSは、IPLによって読み出されます。COMMAND.COMは、MSDOS.SYSによって読み出されます。

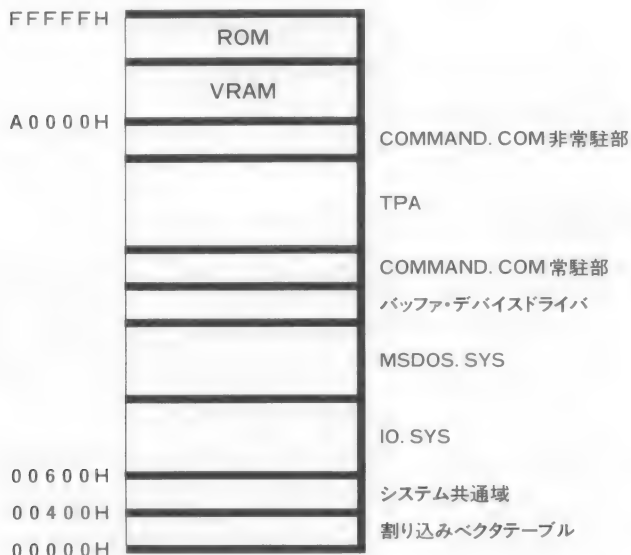
IO.SYSは、主にハードウェアに固有の入出力を受け持ちます。ディスク入出力、キーボード、画面、プリンタなどはここで制御されます。MSDOS.SYSは、MS-DOSの本体でありシステムコールなどを受け持ちます。最後に、COMMAND.COMはユーザの入力するコマンドを解釈して実行する部分で、コマンドプロセッサと呼ばれています。

他には、デバイスドライバや常駐コマンドが結合されている場合もあります。

## ■メモリレイアウト

MS-DOS 起動時のメモリレイアウトは、図 2.2 のようになっています（ただし version 3.10, PS98-127-XXX）。

■図 2.2 MS-DOS のメモリレイアウト



割り込みベクタテーブルの上には、システム共通域と呼ばれる PC-9801 本体の情報が格納されています。ここは特に MS-DOS とは関係ないところです。システム共通域の上には IO.SYS がロードされています。アドレスは 00600H に固定されています。IO.SYS の上には MSDOS.SYS がロードされます。ロードされる位置は IO.SYS のサイズによって異なります。ここで、デバイスドライバの指定があればデバイスドライバがロードされ、バッファの指定

があればバッファが確保されます。

続く領域には、COMMAND.COM がロードされます。この COMMAND.COM は常駐部と非常駐部とに分れており、常駐部が MSDOS.SYS（デバイスドライバ等のある場合にはその上）に続く領域にロードされ、非常駐部は RAM の上限にロードされます。常駐部と非常駐部の中間は TPA（トランジェントプログラムエリア）と呼ばれ、ファイルのかたちで存在するプログラムは、ここにロードされて実行されます。

MS-DOS では、RAM の上限は 9FFFFH に制限されています。これは、MS-DOS を最初に採用した IBM-PC の仕様によっているものなので、PC-9801 でもこれに従っており、動かすことはできません。

A0000H~BFFFFH にはテキスト VRAM とグラフィック VRAM が配置されます。それ以降の領域は ROM となっており、BASIC インタプリタや BIOS が格納されています（実際は拡張用に予約されている部分が多い。また、PC-9801U 以降の機種では拡張グラフィック VRAM が搭載される領域もある）。

## ■システムコール

OS を使用することによる恩恵は、システムコールを利用することができるということにあります。システムコールは、OS のサポートするほとんどの機能をユーザに開放したものであり、ユーザはそれを呼び出すことでファイル管理、メモリ管理などの高度な処理を、単純な手続きで使うことができるのです。

MS-DOS では、システムコールは INT 命令で呼び出すことになっています。MS-DOS の提供するシステムコールとその機能は以下のとおりです。

INT	20H	.....	プログラムの非常駐終了
INT	21H	.....	リクエストサービス
INT	25H	.....	アブソリュートディスクリード
INT	26H	.....	アブソリュートディスクライト
INT	27H	.....	プログラムの常駐終了
INT	29H	.....	特殊なデバイスへの1文字出力

INT 21H については、さらに細かな区分が必要となりますが、詳細については、本書の目的からそのすべてを解説するわけにはいきません。適当な MS-DOS の参考書を参照してください。実際、MS-DOS のシステムコールといえば INT 21H を指します。

## ■プログラムのロードと実行

MS-DOS では、プログラムの実行は、コマンドファイルをシステムコールによってロードすることによって行われます。実際は、ユーザが投入したコマンドを COMMAND.COM が解釈し、必要な制御情報を作成したあとに、コマンドを実行するシステムコールを呼び出すのです。

コマンドの実行時には、PSP（プログラムセグメントプリフィクス）と呼ばれる 256 バイトのブロックが重要な意味を持ちます。PSP は、プログラム実行開始時のレジスタ DS(ES) をセグメントベースとしたオフセット 0000H に配置されています。PSP の構成は、図 2.3 のようになっています。

PSP に含まれる情報の利用については、応用編で触れています。

コマンドは大きく 2 つの形式に分けられ、それぞれ COM モデル、EXE モデルと称されます。COM モデルとは、メモリ上にロード

されるそのままのかたちで、ファイルに納められているコマンドであり、EXE モデルとは、メモリ上にロードされる際の情報を先頭に含んだコマンドのことです。前者は比較的小規模なプログラムに、後者は大規模なプログラムに用いられます。

■図 2.3 P S P

### オフセット

	00	02	04	05	06(※)	0A	0E	
00H	INT20H	最大メモリ セグメント	00H	MSDOS.SYSへのFAR CALL		プログラム終了アドレス	プログラム	
	12		16					
10H	中断アドレス	致命的エラー処理 ルーチンアドレス		システム予約領域				
							2C(※※) 2E	
20H	システム予約領域						環境セグメント アドレス	予約領域
30H	システム予約領域							
40H	システム予約領域							
	50	52	53					5C(※※※)
50H	INT21H	RETF	システム予約領域					第1FCB
								6C(※※※)
60H	第1FCBの続き						第2FCB	
70H	第2FCBの続き							
	80							
80H F0H	文字数	文 字 列				0DH		
	パラメータ領域とデフォルトのDTA(※※※※)							

- ※ オフセット06Hからの1ワードにはセグメント内で有効なバイト数が入る(COMで有効)。
- ※※ 環境を表す文字列の入っている領域のセグメント、オフセットは0。
- ※※※ 2つのFCBは10Hバイトしか離れていないためどちらかが有効。
- ※※※※ DTAとして使うとパラメータの文字列は破壊される。

## ■DEBUG/SYMDEB

DEBUG と SYMDEB は、プログラム解読の際の助けとなる強力なデバッグツールです。基礎編、応用編でもすでに登場していますが、資料編では、これらの起動法や機能について説明します。

### ○起動

起動は、MS-DOS のプロンプトが表示されている状態で、

A>DEBUG [F]

または、

A>SYMDEB [F]

とします。DEBUG は MS-DOS version 2.11 で、SYMDEB は MS-DOS version 3.10 で用いるのがよいでしょう（どのバージョンとの組み合わせも動作します）。

ここから、説明を簡略化するために DEBUG を SYMDEB に含めます。機能的な違いについては、そのつど注釈を加えましょう。

### ○パラメータ

SYMDEB の起動時には、パラメータを与えることができます。このパラメータには、ふだんコマンドを起動するときとまったく同じものを与えます。たとえば、コマンド DUMP を、“README.DOC”というパラメータを与えた状態でデバッグできるようにするには、

A>SYMDEB DUMP.COM README.DOC [F]

とします。気を付けなければならないのは、コマンド名には、必ず

拡張子までを含めた完全なファイル名を与えることです。環境変数 `PATH` に設定してあるからといって省略することはできません。

これでプログラム `DUMP.COM` がメモリ上にロードされ、パラメータとして“`README.DOC`”が与えられた状態になっています。

`SYMDEB` では、パラメータにシンボルマップファイルを与えることもできますが、ここでの目的とは関係がありませんので無視します。

## ○コマンド

`SYMDEB` が起動すると、プロンプトとして“`_`”が表示されています。ここでさまざまなコマンドを入力して、プログラムの実行を追跡し、また解析を行います。コマンドの機能については付録Aを参照してください。

コマンドは、プロンプトが表示されている状態で、

コマンド文字 パラメータ

の形式で入力します。パラメータについては、コマンドごとに書式が異なりますので、同様に付録Aを参照してください。



## 2.3 DISK BASICの構造

DISK BASIC は、PC-9801 本体に標準添付した OS であり、何もしなければ、DISK BASIC が立ち上がるようになっています。ただし、この場合は ROM-BASIC といい、ディスクファイルに関する操作や、日本語処理に関する機能が削除されています。これらの機能を使用するには、DISK BASIC のシステムディスクを使用する必要があります。システムディスクはディスクドライブを内蔵している機種であれば、標準で添付されているはずです。

### ■メモリエイアウト

DISK BASIC 起動時のメモリエイアウトは、図 2.4 のようになっています（ただし、version 4.0, 品番 PC-98H47-MW(K)）。

割り込みベクタの上には、MS-DOS と同様にシステム共通域と呼ばれる PC-9801 本体の情報が格納されています。システム共通域の上には、DISK BASIC のワークエリアが用意されています。

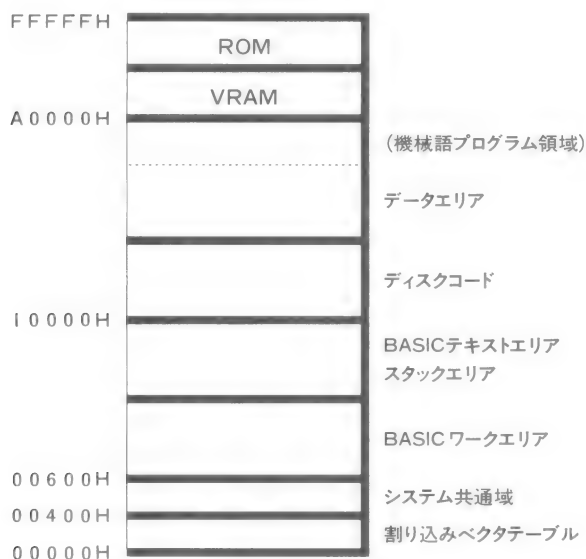
このワークエリアの上には各種 I/O バッファが用意され、その上が DISK BASIC テキストの格納領域、およびシステムスタック領域となります。ここはアドレス 0FFFFH までで、その上にディスクコードと呼ばれるモジュールがロードされます。ディスクコードは、MS-DOS でいう IO.SYS などと同様に IPL によって読み出されるものです。

また、ディスクコードのサイズは、DISK BASIC のバージョンその他の条件によって大きく異なります（ディスクコードが、新たなディスクコードを読み込む場合もある）。

ディスクコードの上にはデータエリアが存在します。DISK BASIC でも RAM の上限は 9FFFFH に制限されています。データエリアと RAM の上限の間には、CLEAR 文によって機械語プログラム領域を設けることができます。

A0000H 以降は、MS-DOS と同様です。

■図 2.4 DISK BASIC のメモリレイアウト



## ■プログラムのロードと実行

DISK BASIC は、OS としての機能と言語としての機能を持った特殊な環境です。言語としての機能は DISK BASIC 言語を解釈し、逐次実行して行くというものです。このとき、DISK BASIC

プログラムはメモリに置かれて、キーボードから打ち込まれるか、LOAD コマンドによってディスクからロードされます。また DISK BASIC プログラムとは別に、機械語のプログラムも存在し、DISK BASIC から CALL 文や USR 関数によって呼び出すことができます。



# 3

## マシン

実際のトリッキーなプログラムは、機械語自体でのアイデアもさることながら、マシンの機能を最大限に利用している場合も多いようです。そこで、マシンについての基礎的な知識が必要となるわけですが、ここでは、そのことについて資料として軽くまとめておきましょう。

## 3.1 割り込み

PC-9801 のサポートする割り込みは多岐にわたり、それぞれが多彩な機能を持っています。また、ハードウェアによって自動的に発生する割り込み（ハードウェア割り込み）と、INT 命令によって発生させる割り込み（ソフトウェア割り込み）があり、これを知っているのと知らないのとでは、解説において大きな差が出ます。

どの割り込みタイプにどのような機能が割り当てられているかは、使用する OS によっても異なりますが、表 3.1 として MS-DOS、DISK BASIC のそれぞれについてまとめておきます。双方で機能が共通の場合は、OS 不使用时にも使用可能な場合が多いようです。

ハードウェアの違い（機種の違い、拡張ボードの有無など）によっても、割り込み機能には差が出ます。

PC-9801 においてサポートされる割り込みの一覧は、付録 B としてまとめてあります。付録 B に示したものはあらゆる OS が動作していないときのものであり、BASIC や MS-DOS が動作すると、新たに割り込みが定義されたり、また内容が変更されたりします。

## 3.2 I/O

割り込みによってサポートされていない機能を使用する場合は、その機能をサポートするハードウェアに対して直接アクセスを行う必要がありますが、そのときに必要なのが I/O に対する知識です。

ふつうハードウェアに対するアクセスは、I/O によって行われるからです。

I/O に関しては OS による違いの出ることはまずなく、違いが出るのは機種種の相違、拡張ボードの有無などです。PC-9801 における I/O の一覧は、付録 C としてまとめてあります。

## 3.3 その他のことから

その他の事項として、重要なことがらをあげておきます。

### ■メモリスイッチ

メモリスイッチは、選択可能な項目に関してユーザが行った設定を保持するためのもので、ソフトウェアによる設定が可能で、かつ電源を落としても、長期にわたってその内容が保持されるというも

のです（不揮発性メモリが使用されている）。メモリスイッチは、アドレス A3FE2H, A3FE6H, A3FEAH, A3FEEH, A3FF2H, A3FF6H の 6 個が設けられており、それぞれは SW1～SW6 と名前が付けられています。

メモリスイッチの機能についての詳細は、本体添付のユーザーズマニュアルを参照することにして、ここではメモリスイッチの読み出し／設定の方法について説明しましょう。

メモリスイッチは不揮発性のメモリですが、通常は読み出しのみが許可されており、書き込みは行っても無視されます。書き込みを行えるようにするには、I/O ポートの 68H に 0DH を出力します。これでメモリスイッチが書き込み許可状態になり、メモリスイッチへの書き込みは有効となります。再び書き込み禁止の状態に戻すには、同じく I/O ポートの 68H に 0CH を出力します。

メモリスイッチの効果的なアクセスは、セグメントベースをテキスト VRAM と同じ、A000H にして行うのがよいでしょう。この場合はメモリスイッチへのアクセスだと気付かずに、テキスト VRAM へのアクセスだと思われる可能性大です。

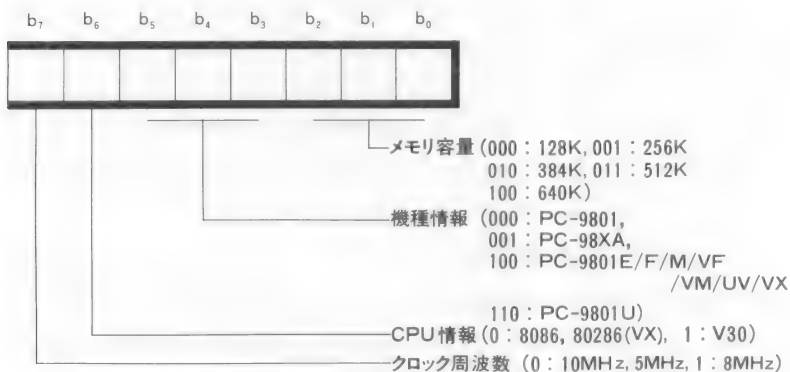
## ■機種・CPU・クロック判別

PC-9800 シリーズの本体（カタログには CPU と記載されている）には、現時点で PC-9801 から PC-9801VX まで、さまざまなバリエーションが存在します。しかし、これらすべてに対応したプログラムを作ったり、また機種を限定して動作させたいときには、機種判別を行わなければなりません。また、CPU によって動作を変更したり、クロックによってタイミングのとり方を変えるなどするときには、それぞれ CPU 判別、クロック周波数の判別を行わなければなりません。そこで、ここでは機種判別、CPU 判別、クロ

ック周波数判別のための情報について説明します。

それぞれの判別を行うには、まずシステム共通域に含まれる 0000H : 0501H の 1 バイトを参照すればよく、機種情報のほか、CPU 情報やクロック周波数の情報が含まれています。このアドレスの構成は、図 3.1 のようになっています。

■図 3.3 0000H : 0501H の構成



まず機種の判別を行い、そこから CPU やクロック周波数の判別を行うのが順序です。機種の判別には、図 3.3 からわかるようにビット 5～3 を見ればよく、この値により PC-9801, PC-98XA, PC-9801E/F/M/VF/VM/UV/VX, PC-9801U の範囲で判別を行うことができます。ここでは、かなりあらっぽい判別しか行えませんが、後述する CPU 情報や、クロック周波数の情報を組み合わせて、最終的な機種を決定します。

機種がわかれば CPU の判別です。判別にはビット 6 を見ます。ここで注意しなくてはならないのは、機種によって CPU とビット値の対応が異なるということです。特に問題となるのは、PC-

9801E/F/M/VF/VM/UV/VX のどれかであるという場合で、PC-9801E/F/M では 8086 のみの搭載、PC-9801VF/VM/UV では V30 のみの搭載、PC-9801VX では、V30 と 80286 の両方を搭載しているということで、1 ビットでは 3 つの情報を表すことができません。そこで、8086 と 80286 の情報を重ねて、同時に 2 つの CPU を表現することにします。ところが、PC-9801E/F/M と PC-9801VX の 80286 モードを判別することはできません。解決については後述します。

次にクロック周波数を判別します。判別にはビット 7 を見ます。クロックが切り替えられる機種の場合、5MHz と 8MHz, 8MHz と 10MHz の 2 通りの切り替え方式を持ちますが、CPU と同様、情報の表現に 1 ビットしか使用できないため、5MHz と 10MHz を兼ねています。しかしこれは問題なく、8086 と V30 を判別すれば、おのずからクロック周波数も限定することができます。

さて、8086 と 80286 の区別ですが、同じ命令でありながら動作が微妙に異なる、という命令を実行させてみればすぐにわかります。これは、PUSH SP 命令を用いればよく、8086 と 80286 で動作が異なるという特性を応用して CPU の判別を行います。具体的には、次のプログラムを参照してください。

```
MOV    AX,SP
PUSH   SP
MOV    BP,SP
CMP    AX, [BP]
POP    SP
```

このプログラムが実行された時点で、ZF = 0 であれば 80286、ZF = 1 であれば 8086 です（V30 も同じ）。なぜこのような違いが



生じるのかというと、8086 ではレジスタ SP の内容をスタックにプッシュしてから、レジスタ SP を減じるのに対し、80286 では、レジスタ SP の内容を減じたあとに、スタックへのプッシュを行います。すなわち、前者ではスタックに積まれている内容とレジスタ SP の内容が異なるのに対し、後者では等しくなり、よって CPU の判別が行えるわけです。

## ■メモリサイズ

メモリサイズを知っておくと、バッファの確保などに役立つ場合があります。メモリサイズにはユーザがメモリスイッチに設定するものと、システムが実際にチェックして、システム共通域に設定するものがあります。どちらを優先すればよいかといわれれば後者を優先したほうがよいでしょう。なぜなら、前者はあくまでもユーザが設定するものであり、メモリ増設を行っていなくても設定を多めにしておいたり、またメモリを減らしたのに設定を忘れている可能性もあるからです。万一メモリに異常があった場合でも、後者ではメモリチェックを行って、正常な部分のみが設定されます。

システムが調査したメモリサイズは、機種情報その他と同時に得ることができます。すなわち 0000H : 0501H を参照し、ビット 2～0 を調べます。値とサイズの関係は、図 3.3 を参照してください。

なおシステム調査によるメモリサイズは、ユーザの設定したメモリサイズを越えて設定されることはありません。

# A

付録  
APPENDIX

## A. SYMDEB機能一覧

ここでは、MS-DOS 上のツールである SYMDEB について、その機能を一覧として示します。なお、一覧中の略語は以下の意味を持ちます。

名前	意 味
SYMBOL	シンボルファイルからのシンボルです。
LINE	行番号を意味します。ソースデバッグを行っているときにのみ意味があります。
NUM	<p>数値を表します。数値には以下に示すサフィックスを付加することで、数値の型を変えることができます。</p> <p>Y     : 2 進数  O, Q : 8 進数  T     : 10進数  H     : 16進数 (省略可)</p> <p>DEBUG では、数値はすべて16進数です。</p>
ADDR	<p>アドレスを表します。アドレスは、</p> <p>セグメント: オフセット</p> <p>の形式か、</p> <p>オフセットアドレス</p> <p>のみの形式で指定します。</p>
RANGE	<p>アドレスの範囲を表します。範囲は、</p> <p>&lt;ADDR&gt; &lt;ADDR&gt;</p> <p>の形式か、</p> <p>&lt;ADDR&gt; L &lt;NUM&gt;</p>
STRING	<p>文字列を表します。文字列は、' か" でくくります。その文字そのものを表すには、文字を 2 個続けます。</p>

名前	意味
EXP	式を表します。式中には、以下の演算子を使用することができます。 単項演算子：+, −, NOT, SEG, OFF, BY, WO, DW, POI, PORT, WPORT 二項演算子：*, /, MOD, :, +, −, AND, XOR, OR
TYPE	型を表します。型には以下の文字を使用することができます。 A：アスキー文字列 B：バイト W：ワード D：ダブルワード S：単精度浮動小数点実数（4バイト長） L：倍精度浮動小数点実数（8バイト長） T：10バイト浮動小数点実数
LIST	数値のリストを表します。
PORT	I/Oアドレスを表します。

書式	機能	
?	コマンド一覧の表示	—
Q	SYMDEBの終了	○
D<TYPE><ADDR> D<TYPE><RANGE>	メモリのダンプ	○
E<TYPE><ADDR><LIST>	メモリ内容の変更	○
I<PORT>	ポートからの入力	○
O<PORT><NUM>	ポートへの出力	○
A<ADDR>	アセンブル	○
U<ADDR> U<RANGE>	逆アセンブル	○
C<RANGE><ADDR>	メモリ内容の比較	○
F<RANGE><LIST>	メモリの充填	○
M<RANGE><ADDR>	メモリ内容のコピー	○
S<RANGE><LIST>	メモリ内容の検索	○
?<EXP>	式の計算	—
H<NUM><NUM>	和と差の計算	○
G=<ADDR><ADDR>	プログラムの実行	○

書 式	機 能	DEBUG
T=<ADDR><NUM>	プログラムのシングルステップ実行	○
P=<ADDR><NUM>	プログラムのシングルステップ実行	—
BP<NUM><ADDR> <NUM><STRING>	ブレークポイントの設定	—
BC<NUM> BC *	ブレークポイントの解除	—
BD<NUM> BD *	ブレークポイントの無効化	—
BE<NUM> BE *	ブレークポイントの有効化	—
R<レジスタ名> R<レジスタ名>=<NUM> RF	レジスタ値の表示 レジスタ値の設定 フラグレジスタの表示・設定	○ — ○
K<NUM>	スタックフレームの表示	—
L<ADDR><ドライブ> <セクタ><セクタ数>	ファイル・セクタの読み込み	○
W<ADDR><ドライブ> <セクタ><セクタ数>	ファイル・セクタの書き込み	○
N<引数>.....	ファイル名・引数の設定	○
!<コマンド>	コマンドの実行	—
< , > , = , ! , ! , -	リダイレクト	—
*<コメント>	コメントの表示	—

## B PC-9801割り込み一覧

PC-9801でサポートされる割り込み（内部、外部含む）の一覧を示します。なお、これらはOSが起動していることを前提としていないため、OSが起動すれば割り込みの種類が増し、変更されることも考えられます。

タイプ番号	ベクタアドレス	機 能	備考
00H	0000H~0003H	除算エラー割り込み	* 1
01H	0004H~0007H	シングルステップ割り込み	* 1
02H	0008H~000BH	マスク不能割り込み (NMI)	
03H	000CH~000FH	トラップ (INT3)	* 1
04H	0010H~0013H	オーバフロー割り込み (INT0)	* 1
05H	0014H~0017H	COPYキー割り込み	
06H	0018H~001BH	STOPキー割り込み	
07H	001CH~001FH	インターバルタイマ	* 1
08H	0020H~0023H	タイマ	
09H	0024H~0027H	キーボード	
0AH	0028H~002BH	CRTV (垂直帰線)	* 2
0BH	002CH~002FH	拡張バス (INT0)	* 2
0CH	0030H~0033H	RS-232C	
0DH	0034H~0037H	拡張バス (INT1, CMT)	* 2
0EH	0038H~003BH	拡張バス (INT2, ODA系プリンタ)	
0FH	003CH~003FH	システム予約	* 2
10H	0040H~0043H	セントロニクス系プリンタ	* 2
11H	0044H~0047H	拡張バス (INT3, ハードディスク)	* 2
12H	0048H~004BH	拡張バス (INT41, 640KBディスク)	
13H	004CH~004FH	拡張バス (INT42, 1MBディスク)	
14H	0050H~0053H	拡張バス (INT5)	* 2
15H	0054H~0057H	拡張バス (INT6)	* 2
16H	0058H~005BH	数値演算プロセッサ (8087)	* 2
17H	005CH~005FH	ノイズ (システム予約)	* 2
18H	0060H~0063H	キーボードBIOS、CRT BIOS	
19H	0064H~0067H	RS-232C BIOS	
1AH	0068H~006BH	CMT BIOS、プリンタBIOS	
1BH	006CH~006FH	ディスクBIOS	
1CH	0070H~0073H	カレンダー、タイマBIOS	
1DH	0074H~0077H	システム予約	* 2
1EH	0078H~007BH	N88-BASIC	
1FH	007CH~007FH	システム予約	
20H~3FH	0080H~00FFH	システム予約	
40H~7FH	0100H~01FFH	ユーザに解放	
80H~FFH	0200H~03FFH	システム予約	

\* 1 IRET 命令へのポインタが設定されている

\* 2 EOI発行ルーチンへのポインタが設定されている

# C PC-9801I/O一覽

PC-9801でサポートされるI/Oの一覧を示します。なおこれらは現時点でのものであり、ハードウェアによっては存在しないもの、または、新たにハードウェアが付け加わることもあり得ます。

ポートアドレス	対応ハードウェア
0 0 0 0 0 × A <sub>0</sub> 0	割り込みコントローラ $\mu$ PD8259A(マスタ)
0 0 0 0 1 × A <sub>0</sub> 0	割り込みコントローラ $\mu$ PD8259A(スレーブ)
0 0 0 A <sub>3</sub> A <sub>2</sub> A <sub>1</sub> A <sub>0</sub> 1	DMAコントローラ $\mu$ PD8237A-5
0 0 1 0 × × × 0	カレンダー時計 $\mu$ PD1990(4990)
0 0 1 0 × A <sub>1</sub> A <sub>0</sub> 0	DMAバンク
0 0 1 1 × × A <sub>0</sub> 0	RS-232Cインタフェース $\mu$ PD8259A
0 0 1 1 × A <sub>1</sub> A <sub>0</sub> 0	システムポート $\mu$ PD8255A-5
0 1 0 0 × A <sub>1</sub> A <sub>0</sub> 0	プリンタインタフェース $\mu$ PD8255A-5
0 1 0 0 × × A <sub>0</sub> 1	キーボードインタフェース $\mu$ PD8251A
0 1 0 1 × × A <sub>0</sub> 0	マスク不能割り込み制御
0 1 0 1 × A <sub>1</sub> A <sub>0</sub> 1	320KBディスクインタフェース $\mu$ PD8255A-5
0 1 1 0 A <sub>2</sub> A <sub>1</sub> A <sub>0</sub> 0	CRTコントローラ $\mu$ PD7220
0 1 1 0 × × × 0	予約
0 1 1 1 A <sub>2</sub> A <sub>1</sub> A <sub>0</sub> 0	CRTコントローラ $\mu$ PD52611
0 1 1 1 × A <sub>1</sub> A <sub>0</sub> 1	タイマコントローラ $\mu$ PD8253-5
1 0 0 0 0 A <sub>1</sub> A <sub>0</sub> 0	固定ディスクインタフェース
1 0 0 0 0 × × 1	予約
1 0 0 0 0 A <sub>1</sub> A <sub>0</sub> 1	BRANCH4670
1 0 0 0 1 A <sub>1</sub> A <sub>0</sub> 0	サウンドボード
1 0 0 1 × A <sub>1</sub> A <sub>0</sub> 0	1MBディスクインタフェース $\mu$ PD765A
1 0 0 1 0 A <sub>1</sub> A <sub>0</sub> 1	CMTインタフェース $\mu$ PD8251A
1 0 0 1 1 0 A <sub>0</sub> 1	GPIBスイッチ
1 0 0 1 1 1 0 1	予約
1 0 1 0 A <sub>2</sub> A <sub>1</sub> A <sub>0</sub> 0	グラフィックコントローラ $\mu$ PD7220
1 0 1 0 A <sub>2</sub> A <sub>1</sub> A <sub>0</sub> 1	文字フォントROM
1 0 1 1 0 A <sub>1</sub> A <sub>0</sub> 0	HDLC/SDLC $\mu$ PD7201
1 0 1 1 1 × × 0	予約
1 0 1 1 × × × 1	HDLC/SDLC $\mu$ PD7253/ $\mu$ PD8255
1 1 0 0 0 A <sub>1</sub> A <sub>0</sub> 0	ODA系プリンタインタフェース $\mu$ PD8255A-5

ポートアドレス	対応ハードウェア
1 1 0 0 1 A <sub>1</sub> A <sub>0</sub> 0	640KB ディスクインタフェース $\mu$ PD765A
1 1 0 0 A <sub>2</sub> A <sub>1</sub> A <sub>0</sub> 1	GPiB インタフェース $\mu$ PD7210
1 1 0 1 $\times$ A <sub>1</sub> A <sub>0</sub> 1	マウスコントロール
1 1 0 1 1 0 1 1	内部サウンド周波数設定
1 1 0 1 1 A <sub>1</sub> A <sub>0</sub> 1	マウス割り込み時間設定
1 1 1 0 0 0 0 0	キーボードインタフェース (スキャン方式)
5	
1 1 1 0 1 1 0 0	

注： $\times$ は、特に値を問わないことを意味する(多くの場合0を設定)。



# D INDEX

## ●あ行

- アドレッシングモード 264
- 暗号化 175
- 異常なファイル 37
- イミディエイトモード 264
- インターバルタイマ割り込み 157
- 演算 176
- 親プロセス 146

## ●か行

- 解読 17
- かくされたファイル 35
- 環境変数 40
- キー 175
- 逆アセンブリリスト 19
- グラフィックVRAM 199
- 子プロセス 146

## ●さ行

- シークレット属性 199
- システムコール 62, 275
- 定石 92
- シングルステップ割り込み 141, 244
- スタック 201
- セグメント 257
- セグメント外CALL 109
- セグメント外RET 113
- セグメントレジスタ 261

## ●た・な行

- タイムアウト割り込み 164
- タイプ 237
- ダミーファイル 129
- チェックルーチン 17
- ディスクBIOS 38
- ディレクトリ 117
- テキスト表示属性 198
- テキストVRAM 195
- ニーモニック 19

## ●は行

- 汎用レジスタ 260
- バンク切り換え 225
- パラメータインタフェース 239
- 比較検討 37
- 不可視属性 31
- 復元する方法 193
- フラグレジスタ 262
- 不良クラスタ 117
- ブートストラップローダ 270
- ブレイクポイント 135
- プリフィクス命令 90
- プリフェッチキュー 221, 266

## ●ま行

- 未定義命令 100
- メモリスイッチ 284
- メモリモード 264
- メモリレイアウト 259

# ●ら・わ行

レジスタモード 264  
ロギング 51  
割り込み 283, 292  
割り込みベクタテーブル 267

# ●A

AAD <N> 101  
AAM <N> 101  
ADD 89  
AF 262  
AUTOEXEC. BAT 34  
AX 260

# ●B

BIU 222  
BP 261  
BX 261

# ●C

CF 262  
CHKEXE1.COM 153  
CHKEXE3.COM 166  
CHKPAR.COM 149  
CHKTIME.COM 80  
CHKTYPE.COM 70  
CHKWAIT.COM 66  
CLRMEM.COM 170  
COMMAND.COM 29, 273  
CONFIG. SYS 34  
CPU 255  
CS 261  
CX 261

# ●D

DEBUG 278  
DEVICE 34  
DF 263  
DI 261  
DIG.COM 35  
DISK BASIC 280  
DISK BASIC のメモリレイアウト 281  
DIV 89  
DLOG.COM 51  
DREG.COM 41  
DS 261  
DX 261

# ●E～G

ES 261  
EU 221  
FAT 117  
FAT.COM 121  
FAT ID 118  
getfat() 119  
GET-PARENT-ENV 148

# ●I～J

IF 263  
IN 86, 115  
INT 命令 103  
INTD0.COM 104  
INT 08H 231  
INT 1BH 38  
INT 1CH 230  
INT 20H 276  
INT 21H 276

INT 25H 276  
 INT 26H 276  
 INT 27H 276  
 INT 29H 276  
 INT 3命令 135  
 INT3.COM 138  
 I/O 284, 294  
 IO.SYS 29, 273  
 IP 262  
 IPL 20, 272  
 JMP <次のアドレス> 85

## ●M~N

MOV CS 101  
 MOVE <REG>, <REG> 85  
 MS-DOS 19, 273  
 MS-DOSのディスクフォーマット 118  
 MS-DOSのメモリアレイアウト 274  
 MSDOS.SYS 29, 273  
 MUL 89  
 NOP 85  
 N88-DISK BASIC 19

## ●O~P

OF 263  
 OS 19, 269  
 OUT 86, 115  
 OVERA.COM 216  
 OVERB.COM 216  
 PF 263  
 POP <REG> 85  
 POP CS 101  
 PROTIME.COM 75

PSP 276  
 PUSH <REG> 85

## ●Q~R

QUEST1.ASM 86  
 QUEST2.ASM 91  
 r/m16 101

## ●S

SF 263  
 SHELL 34  
 SI 261  
 SINGLE.COM 245  
 SINGLE2.COM 249  
 SP 261  
 SS 261  
 SUB 89  
 SYMDEB 278, 290

## ●T~Z

TF 263  
 V30 256  
 XCHG <REG>, <REG> 85  
 XLAT命令 102  
 ZF 263

## ●その他

0クリア 97  
 0テスト 95  
 80286 256  
 8086 255

# 『PC-9800シリーズ ザ・プロテクトII』

## プログラム解読法入門

### 別売ディスクパック版のお知らせ

最新のテクニックの粋を集めてお贈りするプロテクト技術の数々。果して究極のプロテクトは可能か？ 本書では、好評の姉妹編『PC-9800シリーズ ザ・プロテクト』で予告した解読をしにくくするためのテクニックを、初めて公開しました。いくら強いプロテクトでもプログラムを解読され、チェックルーチンはずされてしまったのでは元も子もありません。本書は、この解読プロテクトに関するあらゆる疑問に、できるだけ詳しく解答しました。読者の皆さんにはご理解いただけたかと思います。

しかし、本書にかぎらず、コンピュータの技術を紙面だけで理解するにはどうしても限界があります。そこで、読者のプロテクト技術をより一層確かなものとするために、別売ディスクパック版として、紙の上ではなく本当のプロテクト技術に触れていただくよう、本書に収められたすべてのプログラムと、紙面の都合で掲載しきれなかったプログラムを、一枚のフロッピーディスクに収録しました。本書をさらにご理解いただくために、是非ともご活用ください。

3.5インチ2DD版	定価5800円	送料300円
5 インチ2DD版	定価5800円	送料300円
5 インチ2HD版	定価5800円	送料300円
8 インチ2D 版	定価5800円	送料300円

お求めは、最寄りの書店、マイコンショップ、または直接弊社宛に現金書留にてお願いいたします。なお、現金書留の際にはご希望の書名、製品名、メディア、住所、氏名、電話番号等を明記の上、送料（300円）を添えてお申し込みください。

**秀和システムトレーディング株式会社**

〒107 東京都港区赤坂8-5-29 チェッカービル

姉妹編

# 『PC-9800シリーズ ザ・プロテクト』

## 別売ディスクパック版のお知らせ

既刊「PC-9800シリーズ ザ・プロテクト」に収録されたプロテクトプログラム、およびその応用プログラムを含め、100種類のプロテクトを施したフロッピーディスクが用意されております。本ディスクには、これら 100 種類のプロテクトのすべてをチェックするプログラムが含まれております。このディスクで立ち上げることによって、そのプログラムが起動するようになっています。読者の皆さんは、このディスクの、できるだけ忠実な複製を作るよう挑戦してください。作成した複製をドライブ装置に入れて立ち上げれば、あなたの複製が 100 種類のうち、いくつまでクリアしたかを採点して表示します。

また、本ディスクパック版にはこのプロテクトとは別に、本文中に掲載されたすべてのソースプログラム、およびオブジェクトプログラムが収録されております。これにより皆さんは打ち込む手間も省け、より能率的にご理解いただけるものと思います。

3.5インチ2DD版	定価5800円	送料300円
5 インチ2DD版	定価5800円	送料300円
5 インチ2HD版	定価5800円	送料300円
8 インチ2D 版	定価5800円	送料300円

お求めは、最寄りの書店、マイコンショップ、または直接弊社宛に現金書留にてお願いいたします。なお、現金書留の際にはご希望の書名、製品名、メディア、住所、氏名、電話番号等を明記の上、送料（300円）を添えてお申し込みください。

**秀和システムトレーディング株式会社**

〒107 東京都港区赤坂8-5-29 チェッカービル

**PC-9800シリーズ  
ザ・プロテクト II**  
プログラム解説法入門

---

発行日 1987・3・21 初版第1刷  
1987・11・28 初版第2刷

---

著 者 井上智博・技術開発室©



発行者 牧谷秀昭

発行所 秀和システムトレーディング株式会社  
東京都港区赤坂8-5-29チェッカービル 〒107

印刷所 共同印刷株式会社

---

ISBN4-87966-116-3 C3055 ¥2200E

落丁本・乱丁本はお取り替え致します。なお、本書に関するご質問は小社編集部宛、ご質問の主旨、住所、氏名、電話番号等明記の上、書面にてお願い致します。



秀和システムトレーディング株式会社 定価 2,200円

ISBN4-87966-116-3 C3055 ¥2200E